O'REILLY®

异步图书
www.epubit.com.cn

Python高性能编程

High Performance Python



[美] Micha Gorelick Ian Ozsvald 著
胡世杰 徐旭彬 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

O'REILLY®

Python 高性能编程

[美] Micha Gorelick
Ian Ozsvald 著
胡世杰　徐旭彬　译

人 民 邮 电 出 版 社
北　京

图书在版编目（CIP）数据

Python高性能编程 / (美) 戈雷利克 (Micha Gorelick) , (美) 欧日沃尔德 (Ian Ozsvald) 著 ; 胡世杰, 徐旭彬译. -- 北京 : 人民邮电出版社, 2017.7
ISBN 978-7-115-45489-8

Ⅰ. ①P… Ⅱ. ①戈… ②欧… ③胡… ④徐… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2017)第114298号

版权声明

◆ 著 [美] Micha Gorelick Ian Ozsvald
译 胡世杰 徐旭彬
责任编辑 陈冀康
责任印制 焦志炜
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市海波印务有限公司印刷
◆ 开本：800×1000 1/16
印张：22
字数：415 千字 2017 年 7 月第 1 版
印数：1 – 3 000 册 2017 年 7 月河北第 1 次印刷
著作权合同登记号 图字：01-2013-9303 号

定价：79.00 元

读者服务热线：(010) 81055410 印装质量热线：(010) 81055316
反盗版热线：(010) 81055315

内容提要

Python 语言是一种脚本语言，其应用领域非常广泛，包括数据分析、自然语言处理、机器学习、科学计算、推荐系统构建等。

本书共有 12 章，围绕如何进行代码优化和加快实际应用的运行速度进行详细讲解。本书主要包含以下主题：计算机内部结构的背景知识、列表和元组、字典和集合、迭代器和生成器、矩阵和矢量计算、并发、集群和工作队列等。最后，通过一系列真实案例展现了在应用场景中需要注意的问题。

本书适合初级和中级 Python 程序员、有一定 Python 语言基础想要得到进阶和提高的读者阅读。

前言

Python 很容易学。你之所以阅读本书可能是因为你的代码现在能够正确运行，而你希望它能跑得更快。你可以很轻松地修改代码，反复地实现你的想法，你对这一点很满意。但*能够轻松实现*和*代码跑得够快*之间的取舍却是一个世人皆知且令人惋惜的现象。而这个问题其实是可以解决的。

有些人想要让顺序执行的过程跑得更快。有些人需要利用多核架构、集群，或者图形处理单元的优势来解决他们的问题。有些人需要可伸缩系统在保证可靠性的前提下酌情或根据资金多少处理更多或更少的工作。有些人意识到他们的编程技巧，通常是来自其他语言，可能不如别人的自然。

我们会在本书中覆盖所有这些主题，给出明智的指导去了解瓶颈并提出效率更高、伸缩性更好的解决方案。我们也会在本书中包含那些来自前人的战场故事，让你可以避免重蹈覆辙。

Python 很适合快速开发、生产环境部署，以及可伸缩系统。Python 的生态系统里到处都是帮你解决伸缩性的人，让你有更多时间处理那些更有挑战性的工作。

本书适合哪些人

你使用 Python 的时间已经足够长，了解为什么某些代码会跑得慢，也见过以本书讨论的 Cython、numpy 以及 PyPy 等技术作为解决方案。你可能还有其他语言的编程经验，因此知道解决性能问题的路不止一条。

本书主要的目标读者是那些需要解决 CPU 密集型问题的人，同时我们也会关注数据传输以及内存密集型问题。科学家、工程师、数据分析专家、学者通常会面临这些问题。

我们还会关注网页开发者可能面临的问题，包括数据的移动以及为了快速提升性能而使用 PyPy 这样的即时（JIT）编译器。

如果你有一个 C（或 C++，或 Java）的背景可能会有帮助，但这不是必须的。Python 最常用的解释器（CPython——你在命令行输入 `python` 时启动的标准解释器）是用 C 写的，所以各种钩子和库全都血淋淋地暴露了内部的 C 机制。但我们也会谈到对 C 一无所知的人也能使用的许多其他技术。

你可能还具有 CPU、内存架构和数据总线的底层知识，还是那句话，这些也不完全是必须的。

本书不适合哪些人

本书适用于中高级 Python 程序员。积极的 Python 初学者可能可以跟上，但我们建议要具有坚实的 Python 基础。

我们不会探讨存储系统优化。如果你有一个 SQL 或 NoSQL 瓶颈，本书可能帮不了你。

你会学到什么

我们两位作者在业界和学术界工作了很多年，专门应对大数据应用、处理*我需要更快得到答案！*之类的请求、可伸缩架构等需求。我们会将自己经历千辛万苦获得的经验传授于你，让你免于重蹈覆辙。

在每一章开头，我们会列出问题，并在后续的文字中回答（如果没有回答，告诉我们，我们会在下一个版本中修正！）。

我们会覆盖下面这些主题：

- 计算机内部结构的背景知识，让你知道在底层发生了什么。
- 列表和元组——在这些基本数据结构中细微的语义和速度区别。
- 字典和集合——在这些重要数据结构中的内存分配策略和访问算法。
- 迭代器——Python 风格的代码应该怎样写，用迭代打开无限数据流的大门。
- 纯 Python 方法——如何高效使用 Python 及其模块。
- 使用 `numpy` 的矩阵——像一头野兽一样使用心爱的 `numpy` 库。
- 编译和即时计算——编译成机器码可以跑得更快，让性能分析的结果指引你。
- 并发——高效移动数据的方法。
- `multiprocessing`——使用内建 `multiprocessing` 库进行并行计算的各种方式，高效共享 `numpy` 矩阵、进程间通信（IPC）的代价和收益。
- 集群计算——将你的 `multiprocessing` 代码转换成在研究系统以及生产系统的本地集群或远程集群上运行的代码。
- 使用更少的 RAM——不需要购买大型机就能解决大型问题的方法。
- 现场教训——来自前人的战场故事，让你可以避免重蹈覆辙。

Python 2.7

Python 2.7 在科学和工程计算中是占主导地位的 Python 版本。在*nix 环境(通常是 Linux 或 Mac）下，64 位的版本占了主导地位。64 位让你能够拥有更宽广的 RAM 寻址范围。*nix 让你构建出的应用程序的行为、部署和配置方法都可以很容易地被别人所理解。

如果你是一个 Windows 用户，那么你就要系好安全带了。我们展示的大多数代码都可以正常工作，但有些东西是针对特定操作系统的，你将不得不研究 Windows 下的解决方案。Windows 用户可能面临的最大的困难是模块的安装：搜索 StackOverflow 等站点应该可以帮助你找到你需要的答案。如果你正在使用 Windows，那么使用一台安装了 Linux 的虚拟机（比如 VirtualBox）可能可以帮助你更自由地进行实验。

Windows 用户绝对应该看看那些通过 Anaconda、Canopy、Python(x,y)或 Sage 等 Python 发行版提供的打包的解决方案。这些发行版也会让 Linux 和 Mac 用户的生活简单许多。

迁移至 Python 3

Python 3 是 Python 的未来，每一个人都在迁移过去。尽管 Python 2.7 还将在接下来的很多年里面继续跟我们做伴（有些安装版仍在使用 2004 年的 Python 2.4)，它的退役日期已经被定在 2020 年了。

升级到 Python 3.3+让 Python 库的开发者伤透了脑筋，人们移植代码的速度一直都很慢(这是有原因的)，所以人们转用 Python 3 的速度也很慢。这主要是因为要把一个混用了 Python 2 的 string 和 Unicode 数据类型的应用程序切换成 Python 3 的 Unicode 和 byte 实在太过复杂了。

通常来说，当你需要重现基于一批值得信任的库的结果时，你不会想要站在危险的技术前沿。高性能 Python 的开发者更有可能在接下来的几年里使用和信任 Python 2.7。

本书的大多数代码只需要稍做修改就能运行于 Python 3.3+ (最明显的修改是 `print` 从一个语句变成了一个函数)。在一些地方，我们将特地关注 Python 3.3+带来的性能提升。一个可能需要你关注的地方是在 Python 2.7 中 / 表示 *integer* 的除法，而在 Python 3 中它变成了 *float* 的除法。当然，作为一个好的开发者，你精心编写的单元测试应该已经在测试你的关键代码路径了，所以如果你的代码需要关注这点，那么你应该已经收到来自你单元测试的警告了。

`scipy` 和 `numpy` 从 2010 年开始就已经兼容 Python 3 了。`matplotlib` 从 2012 年开始兼容，`scikit-learn` 是 2013，`NLTK` 是 2014，Django 是 2013。这些库的迁移备忘录可以在它们各自的代码库和新闻组里查看。如果你也有旧代码需要移植到 Python 3，那么就值得回顾一下这些库移植的过程。

我们鼓励你用 Python 3.3+进行新项目的开发，但你要当心那些最近刚刚移植还没有多少用户的库——追踪 bug 将会更困难。比较明智的做法是让你的代码可以兼容 Python 3.3+（学习一下__future__模块的导入），这样未来的升级就会更简单。

有两本参考手册不错：《把 Python 2 的代码移植到 Python 3》和《移植到 Python 3：深度指南》。Anaconda 或 Canopy 这样的发行版让你可以同时运行 Python 2 和 Python 3——这会让你的移植变得简单一些。

版权声明

本书版权符合知识共享协议“署名-非商业性使用-禁止演绎 3.0”。

欢迎以非商业性目的使用本书，包括非商业性教育。本书许可完整转载，如果你需要部分转载，请联系 O’Reilly（见后面“联系我们”部分）。请根据下面的提示进行署名。

我们经过协商认为本书应该使用知识共享许可证，让其内容在世界上更广泛传播。如果这个决定帮到了你，我们将十分高兴收到你的啤酒。我们估计 O’Reilly 的员工对啤酒的看法跟我们相同。

如何引用

如果你需要使用本书，知识共享许可证要求你署名。署名意味着你需要写一些东西让其他人能够找到本书。下面是一个不错的例子：“High Performance Python by Micha Gorelick and Ian Ozsvald (O’Reilly). Copyright 2014 Micha Gorelick and Ian Ozsvald, 978-1-449-36159-4.”

勘误和反馈

我们鼓励你在 Amazon 这样的公开网站上评论本书——请帮助其他人了解他们是否能从本书中受益！你也可以发 E-mail 给我们：feedback@highperformancepython.com。

我们特别希望听到您指出本书的错误，本书帮到你的成功案例，以及我们应该在下一版本加上的高性能技术。你可以通过 O’Reilly 官网给我们留言。

至于抱怨，欢迎你使用即时抱怨传输服务`> /dev/null`。

排版约定

本书采用下列排版约定：

斜体

表示新词、E-mail 地址、文件名，以及文件扩展名。

等宽

用于程序列印，以及在文字中表示命令、模块和程序元素，如变量或函数名、数据库、数据类型、环境变量、语句和关键字。

等宽加粗

表示命令或其他需要用户原封不动输入的文字。

等宽斜体

表示需要被替换成用户指定的值或根据上下文决定的值。



问题

这个记号表示一个问题或练习。



备忘

这个记号表示一个备忘。



警告

这个记号表示一个警告或注意。

使用示例代码

补充材料（示例代码、练习等）可以通过 GitHub 下载。

本书是为了帮你搞定你的问题。通常来说，只要是本书提供的示例代码，你就可以在你的程序和文档中使用。你不需要联系我们获得许可，除非你需要对很大一部分代码进行转载。比如，写一个使用了好几段本书代码的程序不需要许可。以 CD-ROM 的形式销售或分发 O'Reilly 图书中的示例需要许可。引用本书文字和示例代码回答问题不需要许可。在你的产品文档中合并大量本书示例代码需要许可。

如果你觉得你对示例代码的使用超出了上述的许可范围，请通过 permissions@oreilly.com 联系我们。

Safari®在线图书

*Safari 在线图书*是一个应需数字图书馆，它以书和视频的形式提供来自全球顶尖作者的技术和商业内容。

技术专家、软件开发者、网页设计者，以及商业和创新人员将 Safari 在线图书当成他们研究、解决问题、学习和资格认证训练的主要资源。

Safari 在线图书为企业，政府，教育和个人提供了各种收费标准。

其成员可以通过全文搜索数据库访问成千上万的图书，训练视频，以及还未正式出版的手稿。它们来自 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology，以及其他几百个出版社。更多信息请在线访问 https://www.safaribooksonline.com/。

联系我们

请将关于本书的评论和问题发给本书出版社：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

你也可以发送 E-mail 到 bookquestions@oreilly.com 对本书进行评论或询问技术问题。

关于我们的图书、课程、会议和新闻等更多信息请访问我们的网站。

我们的 Facebook：http://facebook.com/oreilly

Twitter：http://twitter.com/oreillymedia

YouTube：http://www.youtube.com/oreillymedia

致谢

感谢来自 Jake Vanderplas、Brian Granger、Dan Foreman-Mackey、Kyran Dale、John Montgomery、Jamie Matthews、Calvin Giles、William Winter、Christian Schou Oxvig、

Balthazar Rouberol、Matt “snakes” Reiferson、Patrick Cooper 和 Michael Skirpan 的反馈和贡献。Ian 感谢他的妻子 Emily 让他消失 10 个月之久来完成本书（她真的太善解人意了）。Micha 感谢 Elaine 及其他朋友和他的家庭能够耐心等待他学习写书。O’Reilly 也是很好的合作伙伴。

第 12 章的提供者非常亲切地共享了他们的时间和宝贵的经验。我们感谢 Ben Jackson、Radim Řehůřek、Sebastjan Trebca、Alex Kelly、Marko Tasic 和 Andrew Godwin 花费的时间和精力。

目录

第 1 章

理解高性能 Python

读完本章之后你将能够回答下列问题

- 计算机架构有哪些元素？
- 常见的计算机架构有哪些？
- 计算机架构在 Python 中的抽象表达是什么？
- 实现高性能 Python 代码的障碍在哪里？
- 性能问题有哪些种类？

计算机编程可以被认为是以特定的方式进行数据的移动和转换来得到某种结果。然而这些操作有时间上的开销。因此，高性能编程可以被认为是通过降低开销（比如撰写更高效的代码）或改变操作方式（比如寻找一种更合适的算法）来让这些操作的代价最小化。

数据的移动发生在实际的硬件上，我们可以通过降低代码开销的方式来了解更多硬件方面的细节。这样的练习看上去可能没什么用，因为 Python 做了很多工作将我们对硬件的直接操作抽象出来。然而，通过理解数据在硬件层面的移动方式以及 Python 在抽象层面移动数据的方式，你会学到一些编写高性能 Python 程序的知识。

1.1 基本的计算机系统

一台计算机的底层组件可被分为三大基本部分：计算单元，存储单元，以及两者之间的连接。除此之外，这些单元还具有多种属性帮助我们了解它们。计算单元有一个属性告诉我们它每秒能够进行多少次计算，存储单元有一个属性告诉我们它能保

存多少数据，还有一个属性告诉我们能以多快的速度对它进行读写，而连接则有一个属性告诉我们它们能以多快的速度将数据从一个地方移动到另一个地方。

通过这些基本单元，我们就可以在各种不同的复杂度级别上描述一个标准工作站。例如，一个标准工作站可以被看作是具有一个中央处理单元（CPU）作为其计算单元，两个独立的存储单元，分别是随机访问内存（RAM）和硬盘（各自有不同的容量和读写速度），最后还有一个总线将所有这些连接在一起。然而，我们还可以深入 CPU 并发现其内部也有多个存储单元：L1、L2，有时甚至有 L3 和 L4 缓存，它们的容量较小（从几 KB 到十几 MB）但速度非常快。这些额外的存储单元通过一个被称为*后端总线*的特殊总线连接 CPU。另外，新的计算机架构通常会有新的配置（比如 Intel 的 Nehalem 架构的 CPU 用英特尔快速通道互联技术替换了前端总线并重新构建了很多连接）。最后，在上述案例的讨论中，我们还忽略了网络连接，这是一种慢速的连接，用于连接其他许多潜在的计算单元和存储单元。

为了帮助理清这些错综复杂的结构，让我们去浏览一下这些基本单元的简要描述。

1.1.1 计算单元

一台计算机的计算单元是其中央部件——它具有将接收到的任意输入转换成输出的能力以及改变当前处理状态的能力。CPU 是最常见的计算单元；然而，最初被设计用于加速计算机图像处理的图形处理单元（GPU）现在变得更加适用于数值计算了，这是因为其自身的并行模式使得大量计算能并发进行。无论哪种类型，一个计算单元都会接收一系列比特（比如代表数字的比特）并输出另外一堆比特（比如代表这些数字之和的比特）。除了实数的基本算数操作和二进制的比特操作以外，一些计算单元还提供了非常特殊的操作，比如“乘加混合计算”，接收三个数字 A、B、C 并返回 A * B + C 的值。

计算单元的主要属性是其每个周期能进行的操作数量以及每秒能完成多少个周期。第一个属性通过每周期完成的指令数（IPC）[①]来衡量，而第二个属性则是通过其时钟速度衡量。当新的计算单元被制造出来时，它们的这两个参数总是互相竞争。比如 Intel 的 Core 系列具有非常高的 IPC 但时钟速度较低，而 Pentium 4 的芯片则完全相反。不过话又说回来，GPU 的 IPC 和时钟速度都很高，但它们有别的问题，我们后面会提到。

另外，当时钟速度提高时，能够立即提高该计算单元上所有的程序运行速度（因为它们每秒能进行更多运算），而提高 IPC 则在矢量计算能力上有相当程度的影响。

① 不要跟进程间通信（也是 IPC）混淆——我们会在第 9 章讨论这个主题。

矢量计算是指一次提供多个数据给一个 CPU 并能同时被操作。这种类型的 CPU 指令被称为 SIMD（单指令多数据）。

总之，计算单元在过去十年的进展颇为有限（图 1-1）。时钟速度和 IPC 的提升都限于停滞，因为晶体管已经小到了物理的极限。结果就是芯片制造商开始依靠其他手段来获得更高的速度，包括超线程技术，更聪明的乱序执行和多核架构。

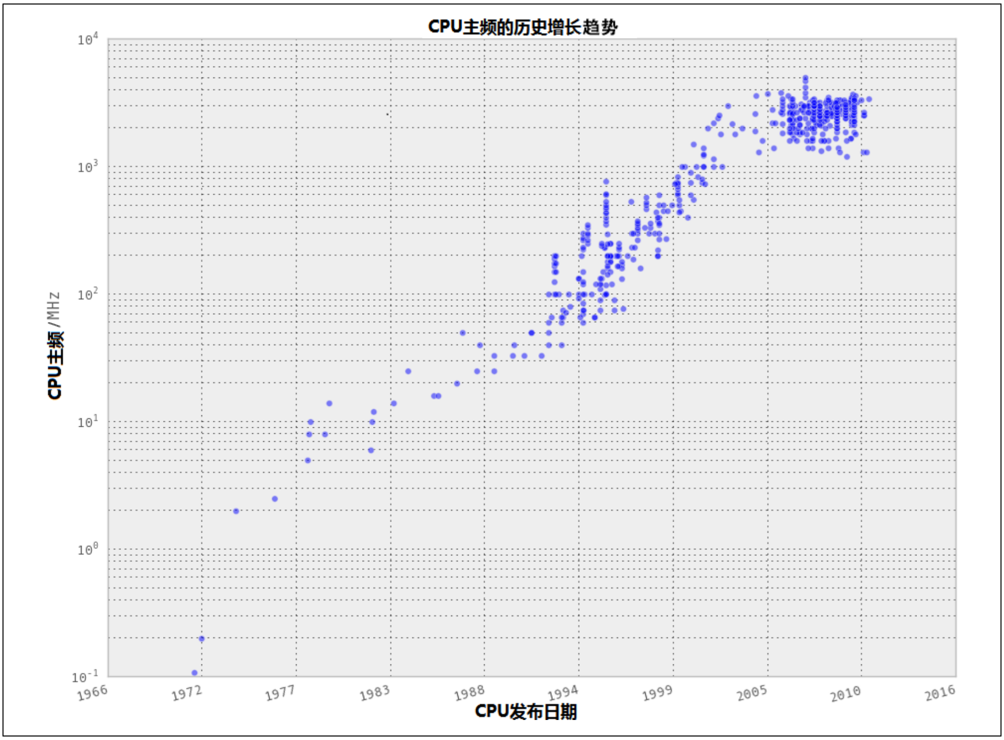


图 1-1　CPU 的时钟速度随时间的变化（数据来自 CPU DB）

超线程技术为主机的操作系统（OS）虚拟了第二个 CPU，而聪明的硬件逻辑则试图将两个指令线程交错地插入单个 CPU 的执行单元。如果成功插入，能比单线程提升 30%。一般来讲，当两个线程的工作分布在不同的执行单元上时，这样做效果不错——比如一个操作浮点而另一个操作整数时。

乱序执行允许编译器检测出一个线性程序中某部分可以不依赖于之前的工作，也就是说两个工作能够以各种顺序执行或同时执行。只要两个工作的成果都能够在正确的时间点上依次得到，哪怕它们的计算次序跟程序设计不同，程序也能继续正确运行。这使得当一些指令被阻塞时（比如等待一次内存访问），另一些指令得以执行，以此来提升资源的利用率。

最后也是对于高级程序员来说最重要的是多核架构的普及。这些架构在同一个计算单元中包含了多个 CPU，提高了总体计算能力，而且无须等待内存屏障，让单个核心可以跑得更快。这就是为什么现在已经很难找到少于双核的计算机了——双核意味着计算机有两个互连的物理计算单元。虽然这增加了每秒可以进行的操作总数，但是想要让两个计算单元都达到最大利用率的话还需要考虑很多错综复杂的因素。

给 CPU 增加更多的核心并不一定能提升程序运行的速度。这是由*阿姆达尔定律*决定的。简单地说，阿姆达尔定律认为如果一个可以运行在多核上的程序有某些执行路径必须运行在单核上，那么这些路径就会成为瓶颈导致最终速度无法通过增加更多核心来提高。

比如，假设我们有一个调查需要 100 个人参与，该调查需要花费 1 分钟，如果我们只有一位提问者（该提问者向第一位参与者提问，等待回答，然后移向下一位参与者），那么我们可以用 100 分钟完成这个任务。这个单人提问并等待回答的流程就是一个顺序执行的过程。对于一个顺序执行的过程，我们每次只能完成一个操作，后面的操作必须等待前面的操作完成。

然而，如果我们有两位提问者，他们就可以同时进行测试，用 50 分钟完成任务。这是由于两位提问者之间不需要互相了解，没有依赖关系，所以整个任务就很容易划分。

增加更多提问者可以进一步提速，直到我们有 100 位提问者。此时，整个流程仅需要 1 分钟就可以完成，仅取决于参与者回答问题所需要的时间。继续增加提问者将不会带来任何速度提升，因为这些多余的提问者无事可干——所有的参与者都已经在接受调查！此时，唯一能够降低整体时间的办法是降低单个参与者完成调查的时间，也就是降低顺序部分所需要的执行时间。同样，对于 CPU，我们可以增加更多的核心直到某个必须单核执行的任务成为瓶颈。也就是说，任何并行计算的瓶颈最终都会落在其顺序执行的那部分任务上。

另外，对于 Python 来说，充分利用多核性能的阻碍主要在于 Python 的*全局解释器锁*（GIL）。GIL 确保 Python 进程一次只能执行一条指令，无论当前有多少个核心。这意味着即使某些 Python 代码可以使用多个核心，在任意时间点仅有一个核心在执行 Python 的指令。以前面调查的例子来说，即使我们有 100 位提问者，然而一次仅有一位可以提问和接受回答，并没有什么用！这看上去是个严重的阻碍，特别是当现在计算机发展的趋势就是拥有更多而非更快的计算单元的时候。好在这个问

题其实可以通过一些方法来避免，比如标准库的 `multiprocessing`，或 `numexpr`、Cython 等技术，或分布式计算模型等。

1.1.2 存储单元

计算机的存储单元被用于保存比特。这些比特可能代表程序中的变量，或一幅图片的像素。存储单元的概念包括了主板上的寄存器、RAM 以及硬盘。所有这些不同类型的存储单元的主要区别在于其读写数据的速度。更复杂的问题在于，其读写数据的速度还与数据的读写方式息息相关。

比如，大多数存储单元一次读一大块数据的性能远好于读多次小块数据（*顺序读取* VS *随机数据*）。将这些存储单元中的数据想象成一本书中的书页，大多数存储单元的读写速度在连续翻页时高于经常从一张随机页跳至另一张随机页。

所有的存储单元或多或少都受到这一影响，但不同类型存储单元受到的影响却大不相同。

除了读写速度以外，存储单元还有一个*延时*的属性，表示了设备为了查找到需要的数据所花费的时间。一个旋转硬盘的延时可能较高，因为磁盘必须物理旋转到一定速度且读取磁头必须移动到正确的位置。而从另一方面来说，RAM 的延时就比较小，因为一切都是固态的。下面是一个标准工作站内常见的各类存储单元的简短描述，以读写速度排序：

旋转硬盘

计算机关机也能保持的长期存储。读写速度通常较慢，因为磁盘必须物理旋转和等待磁头移动。随机访问性能下降但容量很高（TB 级别）。

固态硬盘

类似旋转硬盘，读写速度较快但容量较小（GB 级别）。

RAM

用于保存应用程序的代码和数据（比如用到的各种变量）。具有更快的读写速度且在随机访问时性能良好，但通常受限于容量（GB 级别）。

L1/L2 缓存

极快的读写速度。进入 CPU 的数据必须经过这里。很小的容量（KB 级别）。

图 1-2 展示了当今市面上可以见到的这几类存储单元的区别。

一个清晰可见的趋势是读写速度和容量成反比——当我们试图加快速度时，容量就下降了。因此，很多系统都实现了一个分层的存储：数据一开始都在硬盘里，部分进入 RAM，然后很小的一个子集进入 L1/L2 缓存。这种分层使得程序可以根据访问速度的需求将数据保存在不同的地方。在试图优化程序的存储访问模式时，我们只是简单优化了数据存放的位置、布局（为了增加顺序读取的次数），以及数据在不同位置之间移动的次数。另外，异步 I/O 和缓存预取等技术还提供了很多方法来确保数据在被需要时就已经存在于对应的地方而不需要浪费额外的计算时间——这些过程可以在进行其他计算时独立进行！

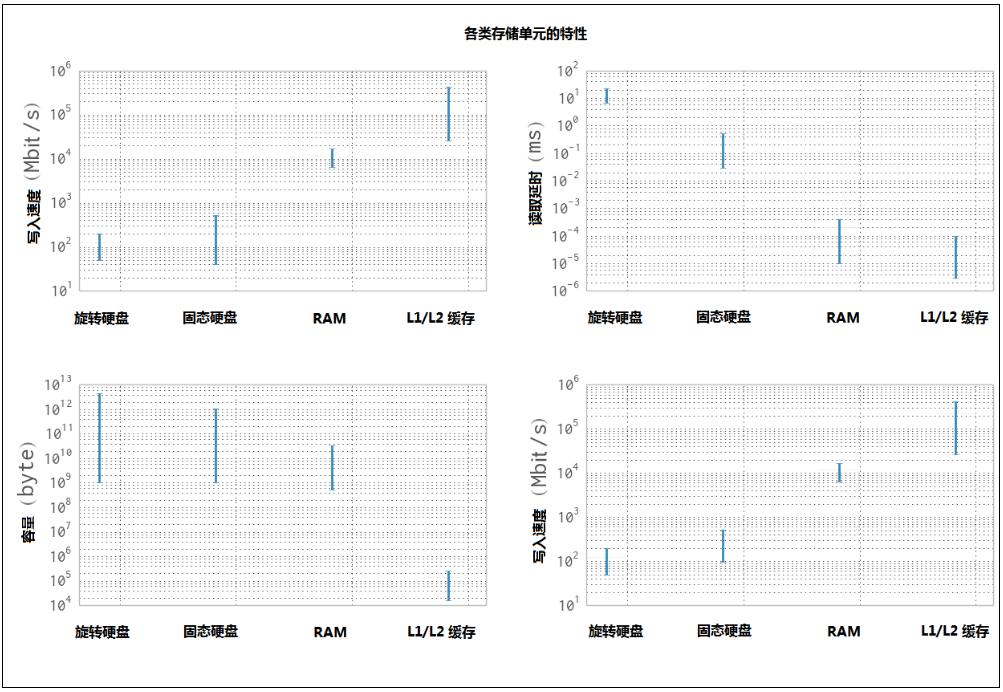


图 1-2　各类存储单元的特征（2014 年 2 月的数据）

1.1.3　通信层

最后，让我们看看这些基本单元如何互相通信。通信有很多模式，但它们都是同一样东西的变种：总线。

比如说，*前端总线*是 RAM 和 L1/L2 缓存之间的连接。它将已经准备好被处理器操作的数据移入一个集结场所以备计算所需，又将计算结果移出。除此之外还有其他

总线，如外部总线就是硬件设备（如硬盘和网卡）通向 CPU 和系统内存的主干线。该总线通常比前端总线慢。

事实上，L1/L2 缓存的很多好处实际上是来自更快的总线。因为可以将需要计算的数据在慢速总线（连接 RAM 和缓存）上攒成大的数据块，然后以非常快的速度从后端总线（连接缓存和 CPU）传入 CPU，这样 CPU 就可以进行更多计算而无须等待这么长的时间。

同样，使用 GPU 的不利之处很多都来自它所连接的总线：因为 GPU 通常是一个外部设备，它通过 PCI 总线通信，速度远远慢于前端总线。结果，GPU 数据的输入输出就像是一种抽税操作。异质架构，一种在前端总线上同时具有 CPU 和 GPU 的计算机架构的兴起就是为了降低数据传输成本，使得 GPU 能够被使用在需要传输大量数据的计算上。

除了计算机内部的通信模块，网络可以被认为是另一种通信模块。不过这个模块就比之前讨论的更为灵活，一个网络设备可以直接连接至一个存储设备，如网络连接存储（NAS）设备或计算机集群中的另一台计算机节点。但是网络通信通常要比之前讨论的其他类型的通信慢很多。前端总线每秒可以传输数十 GB，而网络则仅有数十 MB。

现在我们清楚，一条总线的主要属性是它的速度：在给定时间内它能传输多少数据。该属性由两个因素决定：一次能传输多少数据（总线带宽）和每秒能传输几次（总线频率）。需要说明的是一次传输的数据总是有序的：一块数据先从内存中读出，然后被移动到另一个地方。这就是为什么总线的速度可以被拆分为两个因素，因为这两个因素分别独立影响计算的不同方面：高的总线带宽可以一次性移动所有相关数据，有助于矢量化的代码（或任何顺序读取内存的代码），而另一方面，低带宽高频率有助于那些经常随机读取内存的代码。有意思的是，这些属性是由计算机设计者在主板的物理布局上决定的：当芯片之间相距较近时，它们之间的物理链路就较短，就可以允许更高的传输速度。而物理链路的数量则决定了总线的带宽（带宽这个词真的具有物理上的意义！）。

由于物理接口可以针对某个特定应用优化，所以我们不会奇怪世上存在成百上千种不同类型的连接。图 1-3 显示了一些常见接口的比特率。注意这图上完全没提到连接的延时，延时决定了一个连接响应数据请求花费的时间（虽然延时跟具体的计算机系统息息相关，但是有来自物理接口的一些基本限制）。

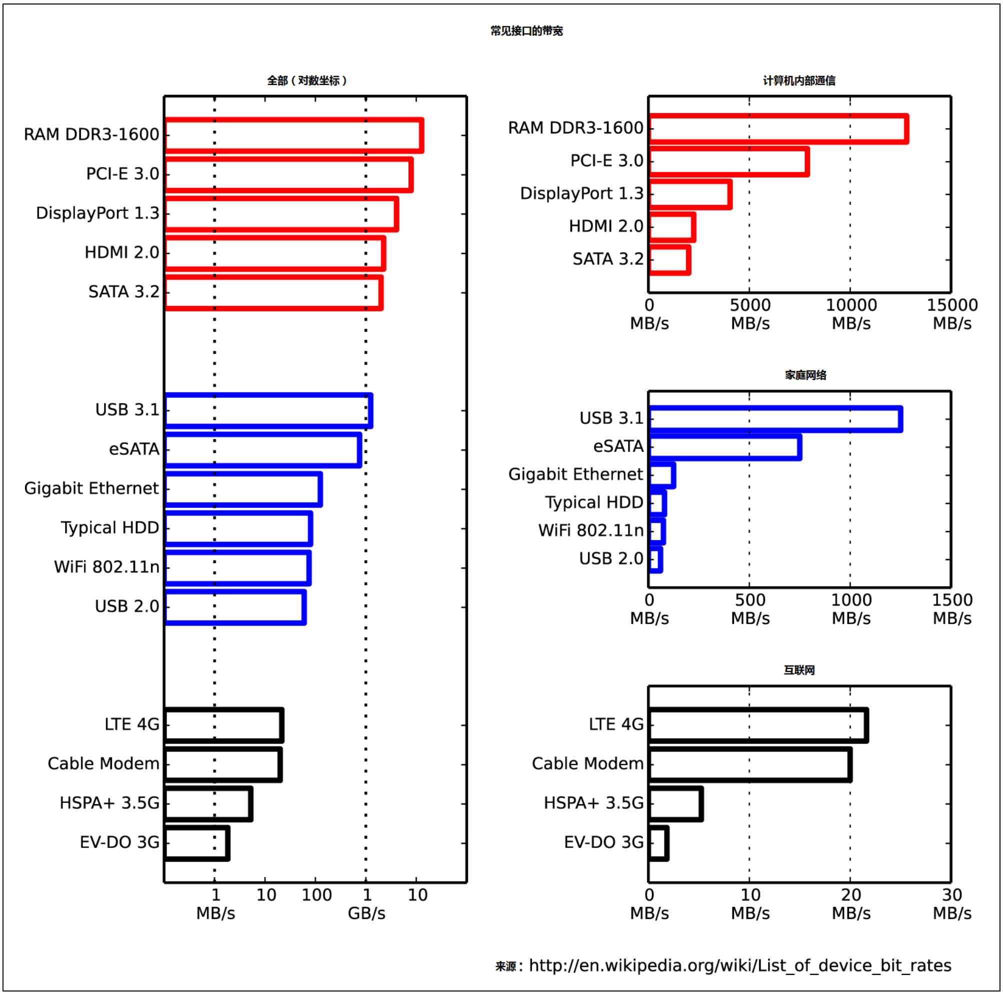


图 1-3　各种常见界面的连接速度（图片来自 Leadbuffalo）

1.2　将基本的元素组装到一起

仅理解计算机的基本组成部分并不足以理解高性能编程的问题。所有这些组件的互动与合作还会引入新的复杂度。本段将研究一些样本问题，描述理想的解决方案以及 Python 如何实现它们。

警告：本段可能看上去让人绝望——大多数问题似乎都证明 Python 并不适合解决性能问题。这不是真的，原因有两点。首先，在所有这些“高性能计算要素”中，我们忽视了一个至关重要的要素：开发者。原生 Python 在性能上欠缺的功能会被

迅速开发出来。另外，我们会在本书中介绍各种模块和原理来帮助减轻这里遇到的问题。当这两点结合起来，我们就能在快速开发 Python 的同时移除很多性能局限。

理想计算模型和 Python 虚拟机

为了更好地理解高性能编程的要素，让我们来看一段用于判断质数的简单代码样例：

```
import math
def check_prime(number):
    sqrt_number = math.sqrt(number)
    number_float = float(number)
    for i in xrange(2, int(sqrt_number)+1):
        if (number_float / i).is_integer():
            return False
    return True

print "check_prime(10000000) = ", check_prime(10000000) # False
print "check_prime(10000019) = ", check_prime(10000019) # True
```

让我们用抽象的计算模型来分析这段代码对比 Python 运行这段代码时实际发生了什么。由于是抽象模型，我们将忽略很多理想化的计算机以及 Python 运行代码方式的细节。不过，这是一个在解决实际问题之前很好的练习：思考算法中的通用组件以及如何最优地使用这些组件来解决问题。只要明白在理想情况下以及实际在 Python 中发生了什么，我们就能让自己的 Python 代码一步步接近最优。

1. 理想计算模型

在代码的开头，我们将 number 的值保存于 RAM 中。为了计算 `sqrt_number` 和 `number_float`，我们需要将该值传入 CPU。在理想情况下，我们只需要传一次，它将被保存在 CPU 的 L1/L2 缓存中，然后 CPU 会进行两次计算并将结果传回 RAM 保存。这是一个理想的情况，因为我们令从 RAM 中读取 number 的值的次数最少，转而以快很多的 L1/L2 缓存的读取代替。另外，我们还令前端总线传输数据的次数最少，以更快的后端总线（连接 CPU 和各类缓存）的传输代替之。将数据保持在需要的地方并尽量少移动这一场景对于优化来说至关重要。所谓“沉重数据”的概念指的是移动数据需要花费时间，而这就是我们需要避免的。

在代码的循环部分，与其一次次将 `i` 输入 CPU，我们更希望一次就将 `number_float` 和多个 `i` 的值输入 CPU 进行检查。这是可能的，因为 CPU 的矢量操作不需要额外的时间代价，意味着它可以一次进行多个独立计算。所以我们希望将 `number_float` 传入 CPU 缓存，以及在缓存放得下的情况下传入尽可能多的 `i` 的值。对于每一对 `number_float/i`，我们将进行除法计算并检查结果是否为整数，

然后传回一个信号表明是否有任意一个结果确实为整数。如果是，则函数结束。如果否，我们继续下一批计算。这样，对于多个 `i` 的值，我们只需要传回一个结果，而不是依靠总线返回所有的值。这利用了 CPU 的矢量化计算的能力，或者说在一个时钟周期内以一条指令操作了多个数据。

这一矢量操作的概念可以用下列代码来表述：

```
import math
def check_prime(number):
    sqrt_number = math.sqrt(number)
    number_float = float(number)
    numbers = range(2, int(sqrt_number)+1)
    for i in xrange(0, len(numbers), 5):
        # the following line is not valid Python code
        result = (number_float / numbers[i:(i+5)]).is_integer()
        if any(result):
            return False
    return True
```

这里，我们让程序一次对 5 个 `i` 的值进行除法和整数检查。当被正确地矢量化时，CPU 仅需一条指令完成这行代码而不是对每个 `i` 进行独立操作。理想情况下，`any(result)`操作将只发生于 CPU 内部而无须将数据传回 RAM。我们将在第 6 章讨论更多关于矢量操作的细节，包括它具体如何工作以及在什么情况下有利于你的代码。

2．Python 虚拟机

Python 解释器为了抽离底层用到的计算元素做了很多工作。这让编程人员无须考虑如何为数组分配内存、如何组织内存以及用什么样的顺序将内存传入 CPU。这是 Python 的一个优势，让你能够集中在算法的实现上。然而它有一个巨大的性能代价。

首先我们要意识到 Python 核心运行于一组非常优化的指令上。而诀窍就是让 Python 以正确的次序执行它们来获得更好的性能。比如下例，我们可以轻松看出，虽然两个算法都有 `O(n)`的运行时间，`search_fast` 会比 `search_slow` 快，因为它提前中止了循环，跳过了不必要的计算。

```
def search_fast(haystack, needle):
    for item in haystack:
        if item == needle:
            return True
    return False

def search_slow(haystack, needle):
    return_value = False
```

```
    for item in haystack:
        if item == needle:
            return_value = True
    return return_value
```

通过性能分析查找代码的慢速区域以及寻找更有效的算法其实就是寻找这些无用的操作并删除它们，最终的结果是一样的，但计算和传输数据的次数却大大减少了。

而 Python 虚拟机抽象层的影响之一就是矢量操作变得不是直接可用了。我们最初的质数函数会循环遍历 `i` 的每一个值而不是将多次遍历组合成一个矢量操作。而我们抽象以后的矢量化代码并不是合法的 Python 代码，因为我们不能用一个列表去除一个浮点。`numpy` 这样的外部库可以通过增加矢量化数学操作的方式帮助我们解决这个问题。

另外，Python 抽象还影响了任何需要为下一次计算保存 L1/L2 缓存中相关数据的优化。这有很多原因，首先是 Python 对象不再是内存中最优化的布局。这是因为 Python 是一种垃圾收集语言——内存会被自动分配并在需要时释放。这会导致内存碎片并影响向 CPU 缓存的传输。另外，我们也没有任何机会去直接改变数据结构在内存中的布局，这意味着总线上的一次传输可能并不包含一次计算的所有相关信息，即使这些信息少于总线带宽。

第二个问题更加基本，根源是 Python 的动态类型以及 Python 并不是一门编译性的语言。很多 C 语言开发者已经在多年开发过程中意识到，编译器总是比你聪明。当编译静态代码时，编译器可以做很多的事情来改变对象的内存布局以及让 CPU 运行某些指令来优化它们。然而，Python 并不是一种编译性的语言：更糟的是，它还具有动态类型，意味着任何算法上可能的优化机会都会更加难以实现，因为代码的功能有可能在运行时被改变。有很多方法可以缓解这一问题，其中最主要的一个方法就是使用 Cython，它可以将 Python 代码进行编译并允许用户“提示”编译器代码究竟有多“动态”。

最后，之前提到的 GIL 会影响并行代码的性能。比如，假设我们改变代码来使用多个 CPU 核心，每个核心收到一堆数字，取值范围是 2 到 `sqrtN`。每个核心可以对自身收到的数据进行计算，当它们都完成时可以互相进行比较。这看上去是一个好方案，虽然我们失去了提前中止循环的能力，但是随着我们使用的核心数的增加，每个核心需要进行的检查数降低了（例如，如果我们有 `M` 个核心，每个核心只需要进行 `sqrtN/M` 次检查）。然而，由于 GIL，一次仅有一个核心可以被使用。这意味着我们还是以非并行的方式运行这段代码，而且还不能提前中止。我们可以使用多进程（`multiprocessing` 模块）而不是多线程，或者使用 Cython 或外部函数来避免这个问题。

1.3 为什么使用 Python

Python 具有高度的表现力且容易上手——新开发者会很快发现他们可以在很短时间里做到很多。许多 Python 库包含了用其他语言编写的工具，使 Python 可以轻易调用其他系统。比如，`scikit-learn` 机器学习系统包含了 LIBLINEAR 和 LIBSVM（两者皆以 C 语言写成），`numpy` 库则包含了 BLAS 以及其他用 C 和 Fortran 语言写的库。因此，正确运用这些库的 Python 代码确实可以在速度上做到跟 C 媲美。

Python 被誉为“内含电池”，因为它内建了很多重要且稳定的库。包括：

unicode 和 *bytes*

语言核心内建。

array

原始类型的高效数组。

math

基本数学操作，包括一些简单的统计数学。

sqlite3

包含了流行的基于文件的 SQL 引擎 SQLite3。

collections

多种对象，包括双向队列、计数器和字典的变种。

除了这些语言核心库，还有大量的外部库，包括：

numpy

一个 Python 数字库（矩阵运算的基石库）。

scipy

大量可信的科学库的集合，通常包含了广受尊重的 C 和 Fortran 库。

pandas

一个数据分析库，类似于 R 语言的数据框或 Excel 表格，基于 `scipy` 和 `numpy`。

scikit-learn

正在快速成为默认的机器学习库，基于 `scipy`。

biopython

一个生物信息学库，类似于 `bioperl`。

tornado

一个提供了并发机制的库。

各类数据库封装

为了跟基本上所有的数据库通信，包括 Redis、MongoDB、HDF5 以及 SQL。

各类网站开发框架

用于创建网站的各种高性能系统，如 `django`、`pyramid`、`flask` 和 `tornado`。

OpenCV

计算机视觉的封装。

各类 *API* 封装

用于轻松访问各种时髦的 web API 如 Google、Twitter 和 LinkedIn。

为了适应各种部署环境，还有大量可选的管理环境和 shell，包括：

- 标准发行版。
- Enthought 公司的 EPD 和 Canopy，一个非常成熟且能干的环境。
- Continuum 公司的 Anaconda，一个注重科学计算的环境。
- Sage，一个类似于 Matlab 的环境，包括一个集成开发环境（IDE）。
- Python(x,y)。
- IPython，一个广泛被科学家和开发人员使用的 Python 互动 shell。
- IPython Notebook，一个基于浏览器的 IPython 前端，广泛用于教学和演示。
- BPython，另一个 Python 互动 shell。

Python 的一大优势在于它可以快速实现出一个新主意的原型。由于存在各种支持库，它能够轻易测试出一个主意是否可行，哪怕第一个实现可能是磕磕碰碰做出来的。

如果你想要让你的数学函数更快，看看 `numpy`。如果你想要实验一下机器学习，试试 `scikit-learn`。如果你在清理和操作数据，那么 `pandas` 是一个好选择。

总的来说，我们有理由提出这样一个问题，“为了让我们的系统跑得更快而进行的优化从长期来看会不会反而导致我们团队整体跑得更慢了？”只要花费足够的人力，系统总是可以被榨出更多的性能，但这可能导致系统脆弱的可维护性以及难以理解的优化并最终导致整个团队绊倒在地。

Cython 就是一个例子（7.6 节），它将 Python 代码注释成类似 C 语言的类型，被转化后的代码可以被一个 C 编译器编译。它在速度上的提升令人惊叹（相对较少的努力就能获得 C 语言的速度），但后续代码的维护成本也会上升。尤其是，对这个新模块可能更难，因为团队成员需要具备一定的编程能力来理解那些为了性能提升而绕开 Python 虚拟机的折衷。

第 2 章

通过性能分析找到瓶颈

读完本章之后你将能够回答下列问题

- 如何找到代码中速度和 RAM 的瓶颈？
- 如何分析 CPU 和内存使用情况？
- 我应该分析到什么深度？
- 如何分析一个长期运行的应用程序？
- 在 CPython 台面下发生了什么？
- 如何在调整性能的同时确保功能的正确？

性能分析帮助我们找到瓶颈，让我们在性能调优方面做到事半功倍。性能调优包括在速度上巨大的提升以及减少资源的占用，也就是说让你的代码能够跑得“足够快”以及“足够瘦”。性能分析能够让你用最小的代价做出最实用的决定。

任何可以测量的资源都可以被分析（不仅是 CPU！）。我们在本章将分析 CPU 的时间和内存的占用。你也可以将同样的技术用于分析网络带宽和磁盘 I/O。

如果一个程序跑得太慢或占用了太多 RAM，那么你一定希望把有问题的代码修正。当然，你完全可以跳过性能分析，修正你认为可能有问题的地方——但是小心，你很有可能“修正了”错误的地方。比起依靠你的直觉，更有效率的做法是先进行性能分析，做出一个假设，然后再改动你的代码结构。

人有时候懒点比较好。先进行性能分析让你能够迅速定位需要被解决的瓶颈，

然后你就可以用最小的改动获得你需要的性能提升。如果你回避性能分析直接进行优化，那么你很有可能最终付出了更多的努力。优化应该总是基于性能分析的结果。

2.1 高效地分析性能

性能分析的首要目标是对受测系统进行测试来发现哪里太慢（或占用太多 RAM，或导致太多磁盘 I/O 或网络 I/O）。性能分析一般会导致额外的性能开销（一般会慢 10 到 100 倍），但你依然希望你的代码尽可能像是在真正的环境中一样运行。所以要以测试用例的方式将你需要测试的那部分系统独立出来。最好这个测试用例已经使用了一套自己的模块。

本章介绍的第一个基本技术包括 IPython 的 `%timeit` 魔法函数，`time.time()`，以及一个计时修饰器。你可以使用这些技术来了解语句和函数的行为。

然后我们会学习 `cProfile`（2.6 节），告诉你如何使用这个内建工具来了解代码中哪些函数耗时最长。这将让你站在高处俯瞰你的问题，使你能够将注意力集中到关键函数上。

接下来，我们会去看 `line_profiler`（2.8 节），这个工具能够对你选定的函数进行逐行分析。其结果将包含每行被调用的次数以及每行花费的时间百分比。这恰能让你知道是哪里跑得慢以及为什么。

有了 `line_profiler` 的结果，你就有了足够的信息去使用编译器（第 7 章）。

在第 6 章（例 6-8），你将学到如何使用 `perf stat` 命令来了解最终执行于 CPU 上的指令的个数以及 CPU 缓存的利用率。这让你能够进一步调优矩阵操作。读完本章后你应该去看看那个例子。

`line_profiler` 之后，我们会演示 `heapy`（2.10 节），它可以追踪 Python 内存中所有的对象——这对于消灭奇怪的内存泄漏特别有用。如果你的系统需要持续运行，那么你会对 `dowser`（2.11 节）感兴趣，它让你能够通过一个 Web 浏览器界面审查一个持续运行的进程中的实时对象。

为了帮助你了解为什么你的 RAM 占用特别高，我们会给你演示 `memory_profiler`（2.9 节）。它能以图的形式展示 RAM 的使用情况随时间的变化，这样你就可以向你的同事们解释为什么某个函数占用了比预期更多的 RAM。

备忘

无论你用什么方法分析代码性能，都必须记得用足够的单元测试覆盖你的代码。单元测试能帮助你避免愚蠢的错误并让你的结果可重现。没有单元测试风险极大。

在编译或重写你的算法之前始终进行性能分析。你需要证据来决定最有效的优化手段。

最后，我们还会给你介绍 CPython 中的 Python 字节码（2.12 节），这样你就能够了解在其台面下发生了什么。具体来说，了解基于栈的 Python 虚拟机如何运行将帮助你明白为什么某个编程风格会跑得比别人慢。

在结束本章之前，我们会回顾如何在性能分析中集成单元测试（2.13 节），让代码跑得更有效的同时维持正确。

最后我们将讨论性能分析的策略（2.14 节），这样你就能够可靠地分析你的代码并收集正确的数据来验证你的假设。在这里，你将了解到动态 CPU 频率以及 TurboBoost 等特性能够如何歪曲你的性能分析结果以及如何禁用这些功能。

为了讲解所有这些步骤，我们需要以一个便于分析的函数为例。下一节我们介绍 Julia 集合。这是一个对 RAM 有一点饥渴的 CPU 密集型函数，而且它还具有非线性的行为（这样我们就无法轻易预测其结果），这意味着我们需要在运行时分析其性能而没法进行线下调查。

2.2 Julia 集合的介绍

让我们从 Julia 集合这个有趣的 CPU 密集型问题开始。这是一个可以产生复杂的输出图像的分形数列，以数学家 Gaston Julia 的名字命名。

函数的代码相当长，你不会想要自己实现一个。它包含一个 CPU 密集型的组件和一个显式的输入集合。这一配置允许我们分析 CPU 和 RAM 的使用情况以帮助我们了解代码中哪部分过多地消耗了这两项计算资源。将代码故意做成非最优的实现，这样我们就可以检查耗内存的操作和慢的语句。我们在本章后面将修正一个慢的语句和一个耗内存的语句，然后在第 7 章，我们还将显著提升整个函数的执行时间。

我们将分析一段能够生成一个伪灰阶图（图 2-1）和一个纯灰阶变种（图 2-3）的代码，设 Julia 集合的复数点 `c=-0.62772-0.42193j`。一个 Julia 集合可以通过

独立计算每一个像素点得到，也就是说这是一个“完美并行计算的问题”，每个点之间没有任何数据共享。

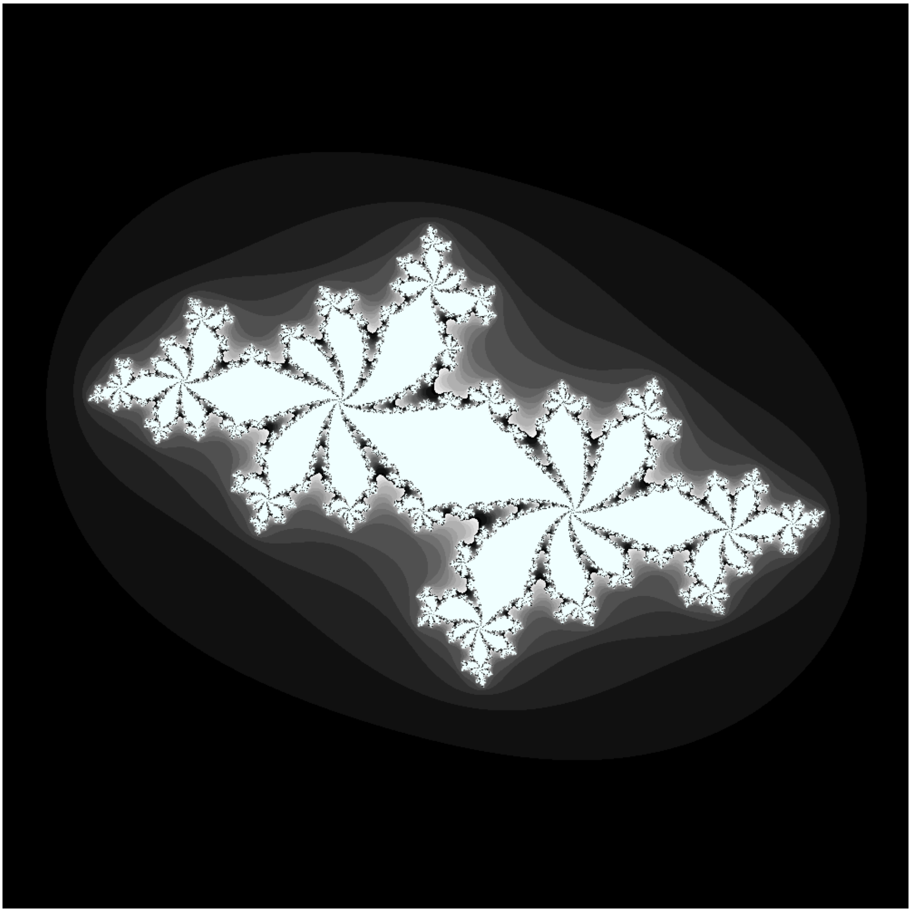


图 2-1　Julia 集合的伪灰阶图，细节高亮

如果我们选择一个不同的 c，就会得到一个不同的图像。根据我们的选择，有些区域算起来较快而另一些会较慢，这有助于我们的分析。

问题的有趣之处在于我们对每一个像素的计算都需要进行一个次数不定的循环。每一次迭代都需要计算坐标值是趋于无穷，还是收敛。那些经过少数迭代就能算出结果的坐标在图 2-1 上为黑色，而那些需要大量迭代才能算出结果的坐标为白色。白色区域需要更多计算，因此生成时间更长。

让我们定义一个 z 的坐标函数进行计算。函数会计算复数 z 的平方加 c：

$$f(z) = z^2 + c$$

我们迭代调用该函数并用 abs 计算逃逸条件。如果逃逸值为 False，那么我们终止循环并在该坐标上记录下迭代的次数。如果逃逸条件始终不满足，那么我们在经过 maxiter 次迭代后终止，并将该 z 坐标转化为一个彩色像素点。

伪代码如下：

```
for z in coordinates:
    for iteration in range(maxiter): # limited iterations per point
        if abs(z) < 2.0: # has the escape condition been broken?
            z = z*z + c
        else:
            break
    # store the iteration count for each z and draw later
```

让我们试算两个坐标来解释这个函数。

首先，我们将使用图的左上角坐标-1.8-1.8j。在坐标更新前我们就必须首先计算逃逸条件 abs(z) < 2：

```
z = -1.8-1.8j
print abs(z)

2.54558441227
```

我们可以看到第 0 次迭代逃逸条件即为 False，于是我们不需要更新坐标。该坐标的输出值就是 0。

现在让我们跳到图中央的 z = 0 + 0j 并尝试几次迭代：

```
c = -0.62772-0.42193j
z = 0+0j
for n in range(9):
    z = z*z + c
    print "{}: z={:33}, abs(z)={:0.2f}, c={}".format(n, z, abs(z), c)

0: z= (-0.62772-0.42193j),              abs(z)=0.76, c=(-0.62772-0.42193j)
1: z= (-0.4117125265+0.1077777992j),    abs(z)=0.43, c=(-0.62772-0.42193j)
2: z=(-0.469828849523-0.510676940018j), abs(z)=0.69, c=(-0.62772-0.42193j)
3: z=(-0.667771789222+0.057931518414j), abs(z)=0.67, c=(-0.62772-0.42193j)
4: z=(-0.185156898345-0.499300067407j), abs(z)=0.53, c=(-0.62772-0.42193j)
5: z=(-0.842737480308-0.237032296351j), abs(z)=0.88, c=(-0.62772-0.42193j)
6: z=(0.026302151203-0.0224179996428j), abs(z)=0.03, c=(-0.62772-0.42193j)
7: z= (-0.62753076355-0.423109283233j), abs(z)=0.76, c=(-0.62772-0.42193j)
8: z=(-0.412946606356+0.109098183144j), abs(z)=0.43, c=(-0.62772-0.42193j)
```

我们可以看到，每次迭代都令 abs(z) < 2 为 True。我们可以对该坐标迭代 300 次依然为 True。我们无法得知需要多少次迭代才能令条件为 False，可能是个无穷数列。最大迭代次数（maxiter）会确保我们不至于永远迭代下去。

我们在图 2-2 中可以看到前 50 个迭代结果。0+0j 的结果数列（带圆形标记的实线）似乎每 8 个迭代出现一次循环，但是每个循环都跟前一个有微小的区别——我们无法得知该坐标是会永远迭代下去，还是将要迭代很长时间，还是马上就会超出边界条件。短划线 cutoff 表示+2 的边界线。

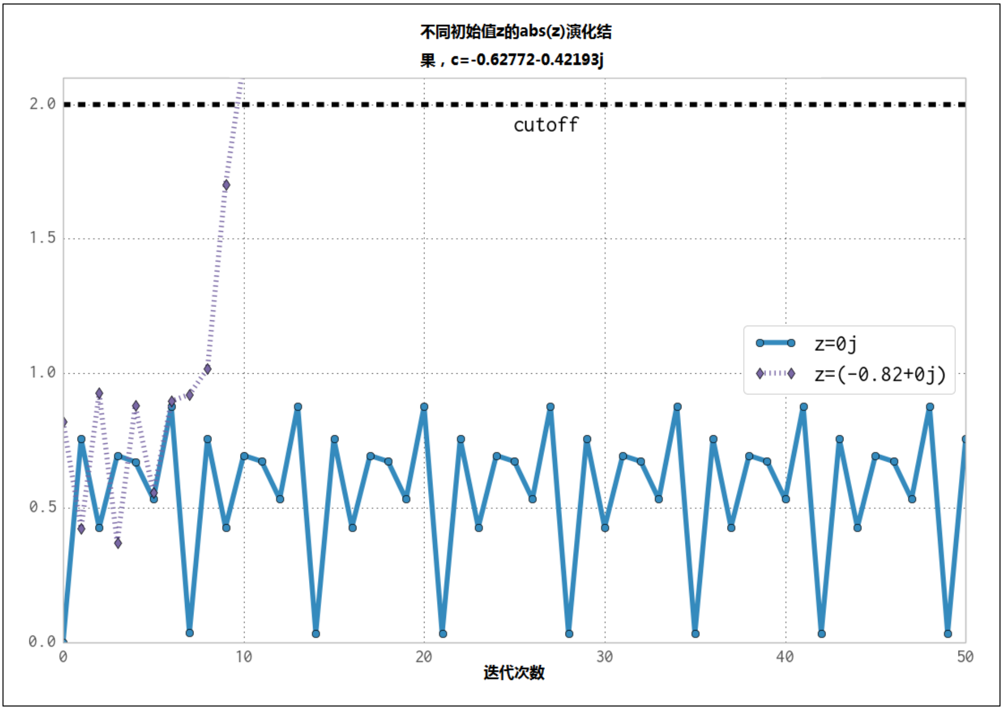


图 2-2　Julia 集合的两个坐标例的演化结果

对于-0.82+0j 的结果数列（带菱形标记的短划线），我们可以看到第 9 次迭代后绝对值结果就超出了+2 的 cutoff 边界线，于是迭代终止。

2.3　计算完整的 Julia 集合

我们在本节分解 Julia 集合的生成代码。我们将在本章以各种方法分析它。如例 2-1 所示，在模块的一开始，我们导入 time 模块作为我们的第一种分析手段并定义一些坐标常量。

例 2-1　定义空间坐标的全局常量

```
"""Julia set generator without optional PIL-based image drawing"""
import time

# area of complex space to investigate
x1, x2, y1, y2 = -1.8, 1.8, -1.8, 1.8
c_real, c_imag = -0.62772, -.42193
```

为了生成图像，我们创建两列输入数据。第一列 zs（复数 *z* 的坐标），第二列 cs（复数的初始条件）。这两列数据都不会发生变化，且 cs 列表可以被优化成一个 c 常量。建立两列输入的理由是之后当我们在本章分析 RAM 使用情况时可以有一些看上去合理的数据来进行分析。

为了创建 zs 列表和 cs 列表，我们需要知道每个 z 的坐标。例 2-2 中，我们用 xcoord 和 ycoord 以及指定的 x_step 和 y_step 创建了这些坐标。这样一个冗长的设定方式有助于将代码移植到其他工具（如 numpy）或 Python 环境上，因为它将一切都明确定义下来，方便除错。

例 2-2　建立坐标列表作为计算函数的输入

```
def calc_pure_python(desired_width, max_iterations):
    """Create a list of complex coordinates (zs) and complex
    parameters (cs), build Julia set, and display"""
    x_step = (float(x2 - x1) / float(desired_width))
    y_step = (float(y1 - y2) / float(desired_width))
    x = []
    y = []
    ycoord = y2
    while ycoord > y1:
        y.append(ycoord)
        ycoord += y_step
    xcoord = x1
    while xcoord < x2:
        x.append(xcoord)
        xcoord += x_step
    # Build a list of coordinates and the initial condition for each cell.
    # Note that our initial condition is a constant and could easily be removed;
    # we use it to simulate a real-world scenario with several inputs to
    # our function.
    zs = []
    cs = []
    for ycoord in y:
        for xcoord in x:
            zs.append(complex(xcoord, ycoord))
            cs.append(complex(c_real, c_imag))
```

```
    print "Length of x:", len(x)
    print "Total elements:", len(zs)
    start_time = time.time()
    output = calculate_z_serial_purepython(max_iterations, zs, cs)
    end_time = time.time()
    secs = end_time - start_time
    print calculate_z_serial_purepython.func_name + " took", secs, "seconds"

    # This sum is expected for a 1000^2 grid with 300 iterations.
    # It catches minor errors we might introduce when we're
    # working on a fixed set of inputs.
    assert sum(output) == 33219980
```

创建了 zs 列表和 cs 列表后，我们输出一些有关列表长度的信息并用 calculate_z_serial_purepython 计算 output 列表。最后我们对 output 的内容求和并断言它符合预期的输出值。Ian 靠它来确保本书此处不至于出现错误。

既然代码已经确定，我们只需要对所有计算出的值求和就可以验证函数是否如我们预期的那样工作。这叫完整性检查——当我们对计算代码进行修改时，很有必要进行这样的检查来确保我们没有破坏算法的正确性。理想情况下，我们会使用单元测试且对该问题的多个不同配置进行测试。

接下来，在例 2-3 中，我们定义 calculate_z_serial_purepython 函数，它扩展了我们之前讨论的算法。注意，我们同时还在函数开头定义了一个 output 列表，其长度跟 zs 和 cs 列表相同。你可能会奇怪为什么我们使用 range 而不是 xrange——这里就先这样，在 2.9 节，我们会告诉你 range 有多浪费。

例 2-3　我们的 CPU 密集型计算函数

```
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while abs(z) < 2 and n < maxiter:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

现在我们可以在例 2-4 中调用整个计算逻辑。只要将它包入一个__main__检查块，我们就可以安全地为某些性能分析手段导入这个模块而不是直接开始计算。注意，将输出画成图像的方法我们这里没有显示出来。

例 2-4　我们代码的 main 函数

```
if __name__ == "__main__":
    # Calculate the Julia set using a pure Python solution with
    # reasonable defaults for a laptop
    calc_pure_python(desired_width=1000, max_iterations=300)
```

一旦我们运行这段代码，就能够看到一些关于问题复杂度的输出：

```
# running the above produces:
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 12.3479790688 seconds
```

图 2-1 所示的伪灰阶图中，高对比的颜色变化给我们一个该函数运行快慢的直观感受。在此处的图 2-3 中，我们有一个线性的灰阶图：黑色是算得快的地方而白色是计算开销大的地方。通过对同一数据的两种表达，我们能够看到线性图上丢失了很多细节。有时，当我们调查一个函数的开销时，多种表达方式有助于我们查清问题。

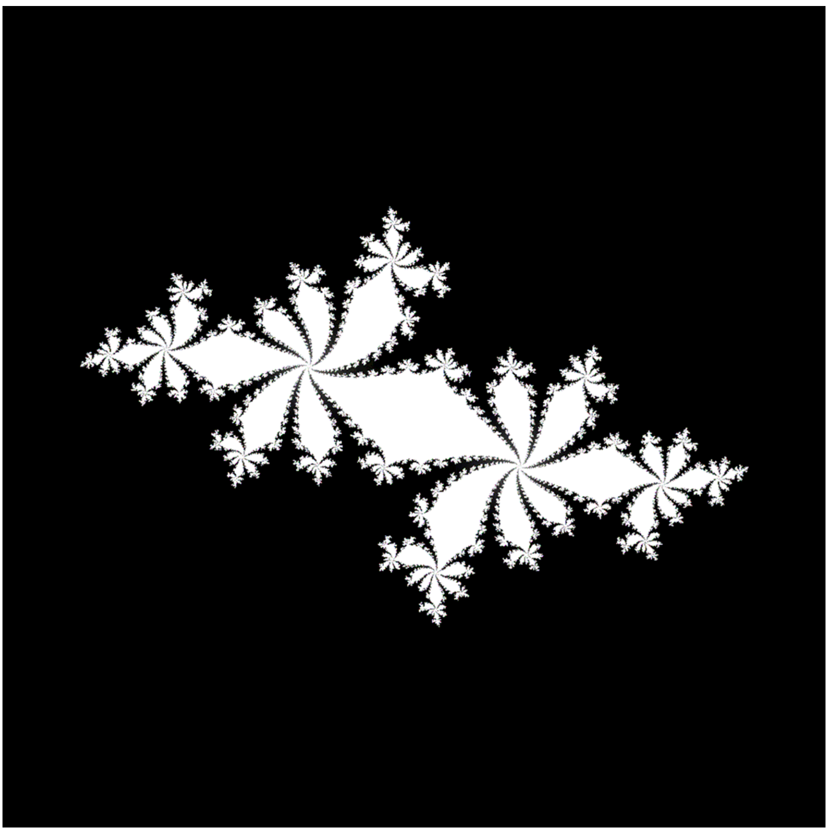


图 2-3　Julia 集合的纯灰阶图

2.4　计时的简单方法——打印和修饰

运行例 2-4，我们看到的输出是由代码中几句 `print` 语句生成的。在 Ian 的笔记本上用 CPython 2.7 跑这段代码要花大约 12 秒。运行时间一般都会有一些波动。你必须在计时的同时观察这些正常的变化，否则你可能会误把某个随机的运行时间的变化当作是由于某次代码的改进造成的。

你的计算机在运行你的代码时还会进行其他任务，比如访问网络、磁盘或 RAM，这些因素都会导致程序运行时间的变化。

Ian 的笔记本是一台 Dell E6420，拥有一个 Intel Core I7-2720QM 的 CPU（2.20GHz，6MB 缓存，4 核）以及 8GB 的 RAM，操作系统是 Ubuntu 13.10。

在 `calc_pure_python` 中（例 2-2），我们能看到一些 print 语句。这是最简单的在函数*内部*测量一段代码执行时间的方法。这个基本方法既快且脏，是在你刚开始着手调查代码时非常有用的手段。

在代码除错和性能分析上使用 `print` 语句是常用的手段。虽然它很快就会变得无法管理，但适用于简短的调查。用完它们以后要记得收拾干净，否则它们会搞乱你的 `stdout`。

一个稍微干净一点的方法是使用修饰器——在需要调查的函数上面增加一行代码。我们的修饰器十分简单，仅仅复制了 `print` 语句的功能。后面我们会让它变得更加高级。

在例 2-5 中，我们定义了一个新函数 `timefn`，它以一个函数 `fn` 为参数。它的内嵌函数 `measure_time` 接受`*args`（数量可变的位置参数）以及`**kwargs`（数量可变的键值对参数）等参数并将其传入 `fn` 执行。在执行 `fn` 前后，我们抓取 `time.time()`并将结果和 `fn.func_name` 一起打印出来。使用这个修饰器的开销很小，但如果你调用上千万次 `fn`，开销就会变得引人注意。我们用`@wraps(fn)`将函数名和 docstring 暴露给 `fn` 的调用者（否则调用者看到的将是修饰器自身的函数名和 docstring，而不是被修饰的函数的）。

例 2-5　定义一个修饰器来自动测量时间

```
from functools import wraps

def timefn(fn):
    @wraps(fn)
    def measure_time(*args, **kwargs):
        t1 = time.time()
```

```
            result = fn(*args, **kwargs)
            t2 = time.time()
            print ("@timefn:" + fn.func_name + " took " + str(t2 - t1) + " seconds")
            return result
        return measure_time


    @timefn
    def calculate_z_serial_purepython(maxiter, zs, cs):
        ...
```

当我们运行这个版本的代码时（之前的 print 语句依然保留），我们会看到修饰器打印的执行时间要略快于 calc_pure_python 打印的时间。这是由于函数的调用带来了额外开销（差异非常小）：

```
Length of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 12.2218790054 seconds
calculate_z_serial_purepython took 12.2219250043 seconds
```

备忘

额外的分析信息不可避免地降低了代码的执行速度——某些性能分析选项非常详细以至于带来了巨大的性能代价。分析信息的细节程度和运行速度是你必须要进行权衡的。

我们可以用 timeit 模块作为另一种测量执行速度的方法。通常来说，你会在解决问题的过程中用它来为各种简单的语句计时。

警告

注意，timeit 模块暂时禁用了垃圾收集器。如果你的操作会调用到垃圾收集器，那么它有可能影响到你实际操作的速度。更多信息请参见 Python 文档：http://bit.ly/timeit_doc。

你可以从命令行运行 timeit 如下：

```
$ python -m timeit -n 5 -r 5 -s "import julia1"
   "julia1.calc_pure_python(desired_width=1000,
   max_iterations=300)"
```

注意你必须以-s 命令在设置阶段导入 julia1 模块，因为 calc_pure_python 来自那个模块。timeit 有一些合理的默认值适用于一段简短的代码，但对于要长期运行的代码来说，最好指定循环次数（-n 5）以及重复次数（-r 5）。timeit 会对语句循环执行 n 次并计算平均值作为一个结果，重复 r 次并选出最好的那个结果。

如果我们不指定-n 和-r 运行 timeit，默认是循环 10 次重复 5 次，需要 6 分钟。改变默认值可以让你更快获得结果。

我们只关注最好的那个结果，平均值以及最差结果可能是由于其他进程的影响。选择循环 5 次重复 5 次应该能给我们一个较为公正的结果：

```
5 loops, best of 5: 13.1 sec per loop
```

试着多运行几次，看看我们会不会得到不同的结果——你可能需要更多的重复次数来获得一个稳定的最佳结果时间。在这一点上不存在“正确”的配置，所以如果你发现你的计时结果变动范围很广，你就要选择更高的重复次数，直到获得稳定的最终结果。

我们的结果显示调用 calc_pure_python 的总体开销是 13.1 秒（最佳情况），而 @timefn 修饰器测算的单次调用 calculate_z_serial_purepython 耗时 12.2 秒。中间的差别主要是用于创建 zs 和 cs 列表的时间。

在 IPython 内部，我们可以用同样的方式使用%timeit 魔法函数。如果你在 IPython 中用互动的方式开发你的代码且函数在本地名字空间(可能是因为你正在使用%run)，那么你可以用：

```
%timeit calc_pure_python(desired_width=1000, max_iterations=300)
```

还有一点值得考虑的是一台计算机上的负载变化。很多后台运行的任务（如 Dropbox、备份等）都会随机影响 CPU 和磁盘资源。网页上的脚本也会导致不可预测的资源使用。图 2-4 显示了我们刚刚进行的计时过程中某个 CPU 的使用率达到了 100%，机器中的其他核心都在轻松处理其他的任务。

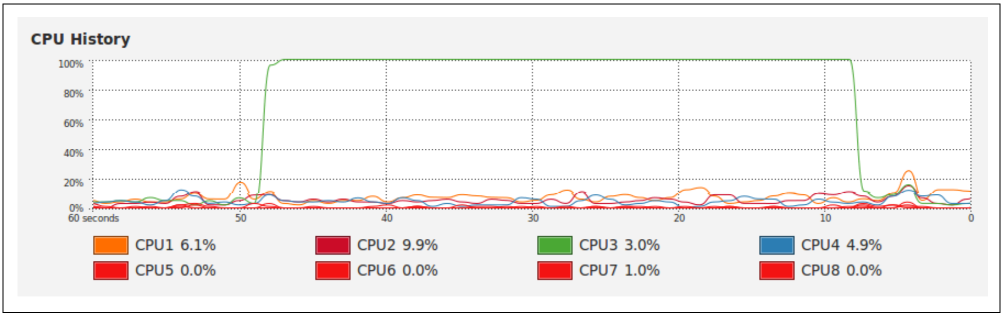


图 2-4　Ubuntu 的系统监视器显示了当我们对程序计时的时候后台 CPU 的使用情况

系统监视器会时不时地显示这台机器上的活动峰值。有必要检查会不会有其他事件发生影响了你的关键资源（CPU、磁盘、网络）。

2.5 用 UNIX 的 time 命令进行简单的计时

现在让我们脱离 Python 使用类 UNIX 操作系统的标准系统功能。下面这条命令会记录程序执行所耗费的的各方面时间，且不在意代码的内部结构：

```
$ /usr/bin/time -p python julia1_nopil.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 12.7298331261 seconds
real 13.46
user 13.40
sys 0.04
```

注意我们特地使用了`/usr/bin/time`而不是`time`，也就是说我们使用的是系统命令的`time`而不是那个更加简单而没用的 shell 内建版本的`time`。如果你用`time --verbose`，结果得到了一个错误，那么你使用的可能就是 shell 内建的`time`而不是系统命令的`time`。

通过使用`-p`开关，我们得到了 3 个结果：

- `real` 记录了整体的耗时。
- `user` 记录了 CPU 花在任务上的时间，但不包括内核函数耗费的时间。
- `sys` 记录了内核函数耗费的时间。

对`user`和`sys`相加就得到了 CPU 总共花费的时间。而这个时间和`real`的差则有可能是花费在等待 IO 上，也可能是你的系统正在忙着运行其他任务因此影响了你的测量。

`time`并不是专为 Python 脚本使用的。它还包括了启动 python 解释器的时间，如果你会启动很多新进程（而不是一个长期运行的单一进程），这个时间可能会很长。如果你会经常跑一些临时脚本，它们的启动时间占了整体运行时间的很大一部分，那么`time`更适合测量这种情况。

我们可以打开`--verbose`开关来获得更多输出信息：

```
$ /usr/bin/time --verbose python julia1_nopil.py
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 12.3145110607 seconds
        Command being timed: "python julia1_nopil.py"
        User time (seconds): 13.46
```

```
System time (seconds): 0.05
Percent of CPU this job got: 99%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:13.53
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 131952
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 58974
Voluntary context switches: 3
Involuntary context switches: 26
Swaps: 0
File system inputs: 0
File system outputs: 1968
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
```

这里最有用的指标可能是 `Major (requiring I/O) page faults`，因为它指示了操作系统是否由于 RAM 中的数据不存在而需要从磁盘上读取页面。而这会带来速度上的惩罚。

我们的例子中对于代码和数据的需求较少，所以不会发生内存缺页错误。如果你有一个内存密集型进程，或多个需要分配和使用大量 RAM 的进程，你就会发现这个命令可以告诉你哪个进程会因为一部分 RAM 被交换到磁盘上这一额外的操作系统级别的磁盘访问而导致速度的下降。

2.6 使用 cProfile 模块

`cProfile` 是一个标准库内建的分析工具。它钩入 CPython 的虚拟机来测量其每一个函数运行所花费的时间。这一技术会引入一个巨大的开销，但你会获得更多的信息。有时这些额外的信息会给你的代码带来令人惊讶的发现。

`cProfile` 是标准库内建的三个分析工具之一，另外两个是 `hotshot` 和 `profile`。`hotshot` 还处于实验阶段，`profile` 则是原始的纯 Python 分析器。`cProfile` 具有跟 `profile` 一样的接口，且是默认的分析工具。如果你对这些库的历史感兴趣，你可以去看一下 Armin Rigo 在 2005 年要求将 `cProfile` 包含进标准库的请求（http://bit.ly/cProfile_request）。

分析工作的一个好的实践是在开始分析之前先对你的代码各部分的运行速度进行假设。Ian 喜欢将有问题的代码打印出来并加以标注。提前生成一个假设意味着有可能测出你错的有多离谱（而且真的会测出！）并提升你对于某种编程风格的直觉。

警告

永远不要忽视靠直觉进行的性能分析（虽然你一定会犯错！）。在分析前先进行假设是绝对值得的，因为这样可以帮助你学习如何定位你代码中可能有问题的地方，而且你应该始终用证据来证明你的选择。

始终基于你的测量结果，用一些既快且脏的分析手段确保你正在分析正确的地方。没有什么比在聪明地优化了一段代码之后（可能过了几小时或几天）才意识到你其实漏掉了进程最慢的部分且根本没有找到真正的问题所在更令人羞愧的了。

那么我们的假设是什么呢？我们知道 `calculate_z_serial_purepython` 可能是代码最慢的部分。我们在那个函数里做了大量的解引用并多次调用基本的算术操作和 `abs` 函数。这些可能都是耗 CPU 资源的大户。

这里，我们用 `cProfile` 模块运行我们的代码的一个变种。其输出是格式化的，方便我们搞清去哪里做进一步分析。

`-s cumulative` 开关告诉 `cProfile` 对每个函数累计花费的时间进行排序，这能让我们看到代码最慢的部分。`cProfile` 会将输出直接打印到屏幕：

```
$ python -m cProfile -s cumulative julia1_nopil.py
...
         36221992 function calls in 19.664 seconds

   Ordered by: cumulative time

   Ncalls tottime percall cumtime percall filename:lineno(function)
        1   0.034   0.034  19.664  19.664 julia1_nopil.py:1(<module>)
        1   0.843   0.843  19.630  19.630 julia1_nopil.py:23
                                           (calc_pure_python)
        1  14.121  14.121  18.627  18.627 julia1_nopil.py:9
                                           (calculate_z_serial_purepython)
 34219980   4.487   0.000   4.487   0.000 {abs}
  2002000   0.150   0.000   0.150   0.000 {method 'append' of 'list' objects}
        1   0.019   0.019   0.019   0.019 {range}
        1   0.010   0.010   0.010   0.010 {sum}
        2   0.000   0.000   0.000   0.000 {time.time}
        4   0.000   0.000   0.000   0.000 {len}
        1   0.000   0.000   0.000   0.000 {method 'disable' of
                                            '_lsprof.Profiler' objects}
```

对累计时间排序能告诉我们大部分的执行时间花在了哪。这个结果显示出在仅仅 19 秒的时间里一共发生了 36 221 992 次函数调用(这个时间包括了使用 `cProfile` 的开销)。之前我们的代码需要 13 秒执行时间——为了测量每个函数执行所花费的时间，我们增加了一个 5 秒的惩罚。

我们可以看到第一行，`julia1_cprofile.py` 的代码入口点总共花费了 19 秒。这是通过`__main__`调用 `calc_pure_python`。`ncalls` 为 `1`,意味着这行仅执行了一次。

在 `calc_pure_python` 内部，`calculate_z_serial_purepython` 的调用花费了 18.6 秒。这两个函数都只执行了一次。我们可以推测出有大约 1 秒的时间花在了 `calc_pure_python` 内部，是调用 `calculate_z_serial_purepython` 以外的代码。不过我们用 `cProfile` 没法知道是哪些行花掉了这些时间。

在 `calculate_z_serial_purepython` 内部,花在(那些没有调用其他函数的)代码行上的时间是 14.1 秒。该函数调用了 34 219 980 次 `abs`，总共花了 4.4 秒，另外还有其他一些不怎么花时间的调用。

那个`{abs}`是什么东西？这一行测量的是 `calculate_z_serial_purepython` 对 `abs` 函数的调用。per-call 的代价可以忽略（0.000 秒），34219980 次调用总共花了 4.4 秒。我们无法预测会调用多少次 `abs`，因为 Julia 函数具有不可预测的动态特性（这也是为什么对它的分析如此有趣）。

我们只能说它最少会被调用 1 000 000 次，因为我们需要计算 `1000×1000` 个像素点。它最多会被调用 300 000 000 次，因为我们对 1 000 000 个像素点最多进行 300 次迭代。所以三千四百万次调用仅是最差情况的 10%。

如果我们看原始的灰阶图（图 2-3），并在想象中将白色部分挤压进角落，我们可以估算出耗时的白色区域大概占了全图的 10%。

接下来的分析输出，`{method 'append' of 'list' objects}`，表示了对 2 002 000 个列表项的创建。

问题

为什么是 2 002 000 个项目？在你读下去之前，先思考一下总共需要创建多少个项目。

2 002 000 个列表项的创建发生在 `calc_pure_python` 的设置阶段。

列表 zs 和 cs 分别有 1000×1000 个项目，而它们是根据 1000 个 x 坐标和 1000 个 y 坐标创建的。所以总共要调用 2 002 000 次 append。

需要注意的是 cProfile 的输出并不以父函数排序，它总结了执行代码块的全部函数。用 cProfile 很难搞清楚函数内的每一行发生了什么，因为我们只拿到了函数自身调用的分析信息，而不是函数内的每一行。

在 calculate_z_serial_purepython 内，我们可以对{abs}和{range}进行分析，这两个函数总计花了大约 4.5 秒。我们知道 calculate_z_serial_purepython 自身总共花费了 18.6 秒。

性能分析输出的最后一行提到了 lsprof，这是本工具进化到 cProfile 之前的原始名字，可以忽略。

为了获得对 cProfile 结果的更多控制，我们可以生成一个统计文件然后通过 Python 进行分析：

```
$ python -m cProfile -o profile.stats julia1.py
```

我们可以这样将其调入 Python，它会输出跟之前一样的累计时间报告：

```
In [1]: import pstats
In [2]: p = pstats.Stats("profile.stats")
In [3]: p.sort_stats("cumulative")
Out[3]: <pstats.Stats instance at 0x177dcf8>
In [4]: p.print_stats()
Tue Jan 7 21:00:56 2014 profile.stats

        36221992 function calls in 19.983 seconds

  Ordered by: cumulative time

  ncalls tottime percall cumtime percall filename:lineno(function)
       1   0.033   0.033  19.983  19.983 julia1_nopil.py:1(<module>)
       1   0.846   0.846  19.950  19.950 julia1_nopil.py:23
                                         (calc_pure_python)
       1  13.585  13.585  18.944  18.944 julia1_nopil.py:9
                                         (calculate_z_serial_purepython)
34219980   5.340   0.000   5.340   0.000 {abs}
 2002000   0.150   0.000   0.150   0.000 {method 'append' of 'list' objects}
       1   0.019   0.019   0.019   0.019 {range}
       1   0.010   0.010   0.010   0.010 {sum}
       2   0.000   0.000   0.000   0.000 {time.time}
       4   0.000   0.000   0.000   0.000 {len}
```

```
        1    0.000    0.000    0.000    0.000 {method 'disable' of
                                                '_lsprof.Profiler' objects}
```

为了追溯我们分析的函数，我们可以打印调用者的信息。在下面的两个列表中，我们可以看到 calculate_z_serial_purepython 是最耗时的函数，且只在一个地方被调用。如果它被多个地方调用，这些列表也许能帮助我们定位到最耗时的父函数：

```
In [5]: p.print_callers()
   Ordered by: cumulative time

Function                                  was called by...
                                              ncalls  tottime  cumtime
julia1_nopil.py:1(<module>)                <-
julia1_nopil.py:23(calc_pure_python)       <-       1    0.846   19.950
                                              julia1_nopil.py:1(<module>)
julia1_nopil.py:9(calculate_z_serial_purepython) <- 1 13.585 18.944
                                              julia1_nopil.py:23
                                              (calc_pure_python)
{abs}                                    <- 34219980    5.340    5.340
                                              julia1_nopil.py:9
                                              (calculate_z_serial_purepython)
{method 'append' of 'list' objects}      <- 2002000    0.150    0.150
                                              julia1_nopil.py:23
                                              (calc_pure_python)
{range}                                  <-       1    0.019    0.019
                                              julia1_nopil.py:9
                                              (calculate_z_serial_purepython)
{sum}                                    <-       1     0.010   0.010
                                              julia1_nopil.py:23
                                              (calc_pure_python)
{time.time}                              <-       2    0.000    0.000
                                              julia1_nopil.py:23
                                              (calc_pure_python)
{len}                                    <-       2    0.000    0.000
                                              julia1_nopil.py:9
                                              (calculate_z_serial_purepython)
                                                  2    0.000    0.000
                                              julia1_nopil.py:23
                                              (calc_pure_python)
{method 'disable' of '_lsprof.Profiler' objects} <-
```

我们还可以反过来显示哪个函数调用了其他函数：

```
In [6]: p.print_callees()
   Ordered by: cumulative time

Function                                 called...
```

```
                                          ncalls  tottime  cumtime
julia1_nopil.py:1(<module>)            ->      1    0.846   19.950
                                          julia1_nopil.py:23
                                          (calc_pure_python)
julia1_nopil.py:23(calc_pure_python)    ->      1   13.585   18.944
                                          julia1_nopil.py:9
                                          (calculate_z_serial_purepython)
                                               2    0.000    0.000
                                          {len}
                                        2002000    0.150    0.150
                                          {method 'append' of 'list'
                                           objects}
                                               1    0.010    0.010
                                          {sum}
                                               2    0.000    0.000
                                          {time.time}
julia1_nopil.py:9(calculate_z_serial_purepython) -> 34219980 5.340 5.340
                                          {abs}
                                               2    0.000    0.000
                                          {len}
                                               1    0.019    0.019
                                          {range}
{abs}                                     ->
{method 'append' of 'list' objects}       ->
{range}                                   ->
{sum}                                     ->
{time.time}                               ->
{len}                                     ->
{method 'disable' of '_lsprof.Profiler' objects} ->
```

cProfile 输出的信息很多，你需要有一个副屏幕才能避免整字换行。它是内建的用于快速定位瓶颈的便利工具。然后我们在本章后面讨论的 line_profiler、heapy 和 memory_profiler 等工具才能帮助你深入定位到需要你加以注意的具体代码行。

2.7 用 runsnakerun 对 cProfile 的输出进行可视化

runsnake 是一个可视化工具，用于显示 cProfile 创建的统计文件——你只需要看它生成的图像就可以快速意识到哪个函数开销最大。

运行 runsnake 可以让你从上层了解一个 cProfile 统计文件的内容，特别是当你在调查一个陌生而又庞大的代码库时。它会让你感觉到应该将注意力集中在哪些区域。它可能会揭示一些你根本就没有意识到会有问题的区域，帮助你定位出潜在的快速优化机会。

你也可以在组内讨论代码性能时使用它，因为它的结果很容易用来讨论。

输入命令 `pip install runsnake` 来安装 `runsnake`。

注意它的安装需要 `wxPython`，而在一个 `virtualenv` 下安装它会非常痛苦。仅仅为了分析一个统计文件，Ian 不止一次地选择宁可在全局环境下安装它，也不愿意尝试让它在 `virtualenv` 下跑起来。

图 2-5 显示了之前 `cProfile` 的数据。图形化的显示可以帮助我们更快地了解 `calculate_z_serial_purepython` 花费了最多的时间，且只有很小一部分执行时间是在调用其他函数（`abs` 是其中唯一一个比较花时间的）。你可以看到你不需要花时间去调查设置阶段，因为大多数执行时间是在计算阶段。

在 `runsnake` 中点击函数可以显示出复杂的嵌套调用。在你跟组员分析性能时，这一功能是无价之宝。



图 2-5　用 runsnakerun 展示 cProfile 的分析结果

2.8　用 line_profiler 进行逐行分析

根据 Ian 的观点，Robert Kern 的 `line_profiler` 是调查 Python 的 CPU 密集型性能问题最强大的工具。它可以对函数进行逐行分析，你应该先用 `cProfile` 找到需要分析的函数，然后用 `line_profiler` 对函数进行分析。

当你修改你的代码时，值得打印出这个工具的输出以及代码的版本，这样你就拥有一个代码变化（无论有没有用）的记录，让你可以随时查阅。当你在进行逐行改变时，不要依赖你的记忆。

输入命令 `pip install line_profiler` 来安装 `line_profiler`。

用修饰器（`@profile`）标记选中的函数。用 `kernprof.py` 脚本运行你的代码，被选函数每一行花费的 CPU 时间以及其他信息就会被记录下来。

备忘

需要修改源代码这点比较讨厌，因为额外的修饰器会影响你的单元测试，除非你创建一个伪修饰器——见 2.13 节中的 No-op 的`@profile`修饰器。

运行时参数`-l` 代表逐行分析而不是逐函数分析，`-v` 用于显示输出。没有`-v`，你会得到一个`.lprof` 的输出文件，回头你可以用 `line_profiler` 模块对其进行分析。例 2-6 中，我们会完整运行一遍我们的 CPU 密集型函数。

例 2-6　运行 kernprof 逐行分析被修饰函数的 CPU 开销

```
$ kernprof -l -v julia1_lineprofiler.py
...
Wrote profile results to julia1_lineprofiler.py.lprof
Timer unit: 1e-06 s

File: julia1_lineprofiler.py
Function: calculate_z_serial_purepython at line 9
Total time: 100.81 s

Line #      Hits   Per Hit  % Time  Line Contents
==================================================
     9                              @profile
    10                              def calculate_z_serial_purepython(maxiter,
                                                                     zs, cs):
11                                      """Calculate output list using
                                        Julia update rule"""
12             1    6870.0     0.0      output = [0] * len(zs)
13       1000001       0.8     0.8      for i in range(len(zs)):
14       1000000       0.8     0.8          n = 0
15       1000000       0.8     0.8          z = zs[i]
16       1000000       0.8     0.8          c = cs[i]
17      34219980       1.1    36.2          while abs(z) < 2 and n < maxiter:
18      33219980       1.0    32.6              z = z * z + c
19      33219980       0.8    27.2              n += 1
20       1000000       0.9     0.9          output[i] = n
21             1       4.0     0.0      return output
```

引入 kernprof.py 导致了额外的运行时间。本例的 calculate_z_serial_purepython 花费了 100 秒，远高于使用 print 语句的 13 秒和 cProfile 的 19 秒。获得的好处则是我们现在得到了一个函数内部每一行花费时间的分析结果。

%Time 列最有用——我们可以看到 36%的时间花在了 while 测试上。不过我们不知道是第一条语句（abs(z) < 2）还是第二条语句（n < maxiter）更花时间。循环内，我们可以看到更新 z 也颇花时间。甚至 n += 1 都很贵！每次循环时，Python 的动态查询机制都在工作，即使每次循环中我们使用的变量都是同样的类型——在这一点上，编译和类型指定（第 7 章）可以给我们带来巨大的好处。创建 output 列表以及第 20 行上的更新相对整个 while 循环来说相当便宜。

对 while 语句更进一步的分析明显就是将两个判断拆开。Python 社区中有一些讨论关于是否需要重写.pyc 文件中对于一行语句中多个部分的具体信息，但目前还没有一个工具提供比 line_profiler 更细粒度的分析。

在例 2-7 中，我们将 while 语句分拆成多个语句。这一额外的复杂度会增加函数的运行时间，因为我们有了更多行代码需要执行，但它可能可以帮助我们了解这部分代码的开销。

问题

在你看代码之前，你是否认为我们可以用这种方式了解基本操作的开销？其他因素会不会让分析变得更复杂？

例 2-7　将组合式 while 语句拆成单个语句来记录每一部分的开销

```
$ kernprof.py -l -v julia1_lineprofiler2.py
...
Wrote profile results to julia1_lineprofiler2.py.lprof
Timer unit: 1e-06 s

File: julia1_lineprofiler2.py
Function: calculate_z_serial_purepython at line 9
Total time: 184.739 s

Line #      Hits     Per Hit   % Time  Line  Contents
===================================================
     9                                 @profile
    10                                 def calculate_z_serial_purepython(maxiter,
                                                                        zs, cs):
    11                                     """Calculate output list using
                                           Julia update rule"""
    12         1    6831.0       0.0       output = [0] * len(zs)
```

```
13    1000001       0.8       0.4      for i in range(len(zs)):
14    1000000       0.8       0.4          n = 0
15    1000000       0.9       0.5          z = zs[i]
16    1000000       0.8       0.4          c = cs[i]
17   34219980       0.8      14.9          while True:
18   34219980       1.0      19.0              not_yet_escaped = abs(z) < 2
19   34219980       0.8      15.5              iterations_left = n < maxiter
20   34219980       0.8      15.1              if not_yet_escaped
                                                   and iterations_left:
21    33219980      1.0      17.5                  z = z * z + c
22    33219980      0.9      15.3                  n += 1
23                                             else:
24    1000000       0.8       0.4                  break
25    1000000       0.9       0.5              output[i] = n
26          1       5.0       0.0          return output
```

这个版本花了 184 秒执行，而之前的仅 100 秒。其他因素*确实*让分析变得更复杂。本例中每一条额外语句都执行了 34219980 次，拖慢了代码。如果不是通过 kernprof.py 调查了每行的影响，我们可能会在缺乏证据的情况下得出是其他原因导致了变慢的结论。

此时有必要回到之前的 timeit 技术来测试每个单独表达式的开销：

```
>>> z = 0+0j # a point in the middle of our image
>>> %timeit abs(z) < 2 # tested inside IPython

10000000 loops, best of 3: 119 ns per loop

>>> n = 1
>>> maxiter = 300
>>> %timeit n < maxiter

10000000 loops, best of 3: 77 ns per loop
```

从这一简单分析上来看，对 n 的逻辑测试的速度几乎是 abs 函数调用的两倍。既然 Python 语句的评估次序是从左到右且支持短路，那么我们应该将最便宜的测试放在左边。每 301 次测试就有 1 次 n < maxiter 的值为 False，这样 Python 就不必评估 and 操作符右边的语句了。

在评估前我们永远无法知道 abs(z) < 2 的值何时为 False，而我们之前对复数平面的观察告诉我们 300 次迭代中大约 10%的可能是 True。如果我们想要更进一步了解这段代码的时间复杂度，有必要继续进行数值分析。不过在目前的情况下，我们只是想要看看有没有快速提高的机会。

我们可以做一个新的假设声明，“通过交换 while 语句的次序，我们会获得一个可

靠的速度提升。”我们可以用 kernprof.py 测试这个假设，但是其额外的开销可能会给我们的结果带来太多噪声。所以我们用一个之前版本的代码，测试比较 while abs(z) < 2 and n < maxiter:和 while n < maxiter and abs(z) < 2:之间的区别。

结果显示出大约 0.4 秒的稳定提升。这一结果显然很无足轻重且局限性太强，使用另一个更合适的方法（如换用第 7 章描述的 Cython 或 PyPy）来解决问题会带来更高的收益。

我们对自己的结果有信心，是因为：

- 我们声明的假设易于测试。
- 我们对代码的改动仅局限于假设的测试（永远不要一次测试两件事!）。
- 我们收集了足够的证据支持我们的结论。

为了保持完整性，我们可以在包含了我们优化的两个主要函数上最后运行一次 kernprof.py 来确认我们代码整体的复杂度。例 2-8 交换了第 17 行 while 测试的语句，我们可以看到原来占用的 36.1%的执行时间现在仅占用 35.9%（这一结果在多次运行中稳定存在）。

例 2-8　交换 while 语句的次序提升测试的速度

```
$ kernprof.py -l -v julia1_lineprofiler3.py
...
Wrote profile results to julia1_lineprofiler3.py.lprof
Timer unit: 1e-06 s

File: julia1_lineprofiler3.py
Function: calculate_z_serial_purepython at line 9
Total time: 99.7097 s

Line #      Hits   PerHit   %Time   Line Contents
==================================================
     9                              @profile
    10                              def calculate_z_serial_purepython(maxiter,
                                                                   zs, cs):
    11                                  """Calculate output list using
                                        Julia update rule"""
    12         1   6831.0     0.0       output = [0] * len(zs)
    13   1000001      0.8     0.8       for i in range(len(zs)):
    14   1000000      0.8     0.8           n = 0
    15   1000000      0.9     0.9           z = zs[i]
    16   1000000      0.8     0.8           c = cs[i]
    17  34219980      1.0    35.9           while n < maxiter and abs(z) < 2:
```

```
    18 33219980      1.0    32.0              z = z * z + c
    19 33219980      0.8    27.9              n += 1
    20  1000000      0.9    0.9           output[i] = n
    21    1          5.0    0.0       return output
```

和预期的一样，我们可以看例 2-9 的输出中，`calculate_z_serial_purepython` 占用了其父函数 97%的时间。创建列表的步骤相对来说无足轻重。

例 2-9　逐行测试设置阶段的开销

```
File: julia1_lineprofiler3.py
Function: calc_pure_python at line 24
Total time: 195.218 s

Line #     Hits   Per Hit   % Time  Line Contents
=================================================
    24                              @profile
    25                              def calc_pure_python(draw_output,
                                                         desired_width,
                                                         max_iterations):
...
    44         1       1.0      0.0     zs = []
    45         1       1.0      0.0     cs = []
    46      1001       1.1      0.0     for ycoord in y:
    47   1001000       1.1      0.5         for xcoord in x:
    48   1000000       1.5      0.8             zs.append(
                                                   complex(xcoord, ycoord))
    49   1000000       1.6      0.8             cs.append(
                                                   complex(c_real, c_imag))
    50
    51         1      51.0      0.0      print "Length of x:", len(x)
    52         1      11.0      0.0      print "Total elements:", len(zs)
    53         1       6.0      0.0      start_time = time.time()
    54         1 191031307.0      97.9   output =
                                           calculate_z_serial_purepython
                                           (max_iterations, zs, cs)
    55         1       4.0      0.0      end_time = time.time()
    56         1       2.0      0.0      secs = end_time - start_time
    57         1      58.0      0.0      print calculate_z_serial_purepython
                                           .func_name + " took", secs, "seconds"
    58
                                       # this sum is expected for 1000^2 grid...
    59         1    9799.0      0.0    assert sum(output) == 33219980
```

2.9　用 memory_profiler 诊断内存的用量

和 Rober Kern 实现的 `line_profiler` 包测量 CPU 占用率类似，Fabian Pedregosa 和 Philippe Gervais 实现的 `memory_profiler` 模块能够逐行测量内存占用率。了解代码的内存使用情况允许你问自己两个问题：

- 我们能不能重写这个函数让它使用更少的 RAM 来工作得更有效率？
- 我们能不能使用更多 RAM 缓存来节省 CPU 周期？

`memory_profiler` 的操作和 `line_profiler` 十分类似，但是运行速度要慢的多。如果你安装了 `psutil` 包（强烈推荐），`memory_profiler` 会跑得快一点。内存分析可以轻易让你的代码慢上 10 到 100 倍。所以实际操作时你可能只是偶尔使用 `memory_profiler` 而更多地使用 `line_profiler` 来进行 CPU 分析。

用命令 `pip install memory_profiler` 来安装 `memory_profiler`（可选装 `pip install psutil`）。

之前说过，`memory_profiler` 的实现可能并不如 `line_profiler` 那么有效率。所以你最好将测试局限在一个较小的问题上，这样才能在一个可容忍的时间内结束。最终验证可以用一整夜来跑，但你需要一个迅速而合理的迭代周期用来分析问题并验证假设。例 2-10 的代码使用了完整的 1000×1000 网格，在 Ian 的笔记本电脑上花了 1.5 个小时来收集数据。

备忘

需要修改源代码这点比较讨厌。和 `line_profiler` 一样，修饰器（`@profile`）被用来标记选中的函数。这会影响你的单元测试，除非你创建一个伪修饰器——见第 57 页 No-op 的`@profile` 修饰器。

在处理内存分配时，你必须意识到情况不像 CPU 占用率那么直截了当。通常让一个进程将内存超额分配给本地内存池并在空闲时使用会更有效率，因为内存分配操作非常昂贵。另外，垃圾收集不会立即进行，所以对象可能在被销毁后依然存在于垃圾收集池中一段时间。

使用这些技术的后果就是很难真正了解一个 Python 程序内部的内存使用和释放的情况，因为*当从进程外部观察时*，某一行代码可能不会分配固定数量的内存。观察多行代码的内存占用趋势可能比只观察一行代码更具有洞察力。

让我们看看 `memory_profiler` 在例 2-10 中的输出。在第 12 行的 `calculate_z_serial_purepython` 中，我们看到分配 1000000 个项目导致大约 7MB 的 RAM[1]被加入这个进程。这不意味着 output 列表的大小就是 7MB，只是进程在列表内部分配时增长了大约 7MB。第 13 行，我们看到进程在循环内又增长了 32MB。

① `Memory_profiler` 测量内存的单位是国际电工委员会的 MiB（mebibyte）为 2^20 字节，这跟更广为人知但也更有歧义的 MB（megabyte 同时有两个广为人知的定义）有轻微的区别。1 MiB 等于 1.048576（大约 1.05）MB。为了讨论方便，除非特别指明，我们将两者视作相等。

这可能是由于调用了 `range`。（例 11-1 进一步讨论了内存追溯，7MB 和 32MB 的区别源于两个列表的内容。）在第 46 行的父进程中，我们看到 `zs` 和 `cs` 列表的分配占用了大约 79MB。再强调一遍，这个数字不一定是数组的真实大小，只是进程在创建这些列表的过程中增长的大小。

例 2-10　memory_profiler 对我们两个主要函数的分析结果，在 calculate_z_serial_purepython 上显示出预料外的内存使用

```
$ python -m memory_profiler julia1_memoryprofiler.py
...
Line#    Mem usage    Increment   Line Contents
================================================
     9   89.934 MiB    0.000 MiB   @profile
    10                             def calculate_z_serial_purepython(maxiter,
                                                                     zs, cs):

    11                                 """Calculate output list using...
    12   97.566 MiB    7.633 MiB       output = [0] * len(zs)
    13  130.215 MiB   32.648 MiB       for i in range(len(zs)):
    14  130.215 MiB    0.000 MiB           n = 0
    15  130.215 MiB    0.000 MiB           z = zs[i]
    16  130.215 MiB    0.000 MiB           c = cs[i]
    17  130.215 MiB    0.000 MiB           while n < maxiter and abs(z) < 2:
    18  130.215 MiB    0.000 MiB               z = z * z + c
    19  130.215 MiB    0.000 MiB               n += 1
    20  130.215 MiB    0.000 MiB           output[i] = n
    21  122.582 MiB   -7.633 MiB       return output

Line #    Mem usage    Increment   Line Contents
================================================
    24   10.574 MiB -112.008 MiB   @profile
    25                             def calc_pure_python(draw_output,
                                                        desired_width,
                                                        max_iterations):
    26                                 """Create a list of complex ...
    27   10.574 MiB    0.000 MiB       x_step = (float(x2 - x1) / ...
    28   10.574 MiB    0.000 MiB       y_step = (float(y1 - y2) / ...
    29   10.574 MiB    0.000 MiB       x = []
    30   10.574 MiB    0.000 MiB       y = []
    31   10.574 MiB    0.000 MiB       ycoord = y2
    32   10.574 MiB    0.000 MiB       while ycoord > y1:
    33   10.574 MiB    0.000 MiB           y.append(ycoord)
    34   10.574 MiB    0.000 MiB           ycoord += y_step
    35   10.574 MiB    0.000 MiB       xcoord = x1
    36   10.582 MiB    0.008 MiB       while xcoord < x2:
    37   10.582 MiB    0.000 MiB           x.append(xcoord)
    38   10.582 MiB    0.000 MiB           xcoord += x_step
    ...
```

```
44   10.582 MiB    0.000 MiB       zs = []
45   10.582 MiB    0.000 MiB       cs = []
46   89.926 MiB   79.344 MiB       for ycoord in y:
47   89.926 MiB    0.000 MiB          for xcoord in x:
48   89.926 MiB    0.000 MiB              zs.append(complex(xcoord, ycoord))
49   89.926 MiB    0.000 MiB              cs.append(complex(c_real, c_imag))
50
51   89.934 MiB    0.008 MiB        print "Length of x:", len(x)
52   89.934 MiB    0.000 MiB        print "Total elements:", len(zs)
53   89.934 MiB    0.000 MiB        start_time = time.time()
54                                  output = calculate_z_serial...
55  122.582 MiB   32.648 MiB        end_time = time.time()
...
```

另一种展示内存使用变化的方式是随时间采样并画图。memory_profiler 有一个功能叫 mprof，用于对内存使用情况进行采样和画图。它的采样基于时间而不是代码行，因而不会影响代码的运行时间。

图 2-6 是 mprof 运行 julia1_memoryprofiler.py 生成的。它会首先生成一个统计文件，然后再用 mprof 画图。图中展示了我们的两个主要函数的执行开始时间以及运行时 RAM 的增长情况。在 calculate_z_serial_purepython 内，我们可以看到 RAM 在函数的整个执行时间内都平稳增长，这是为了生成那些小对象（int 和 float 类型）。

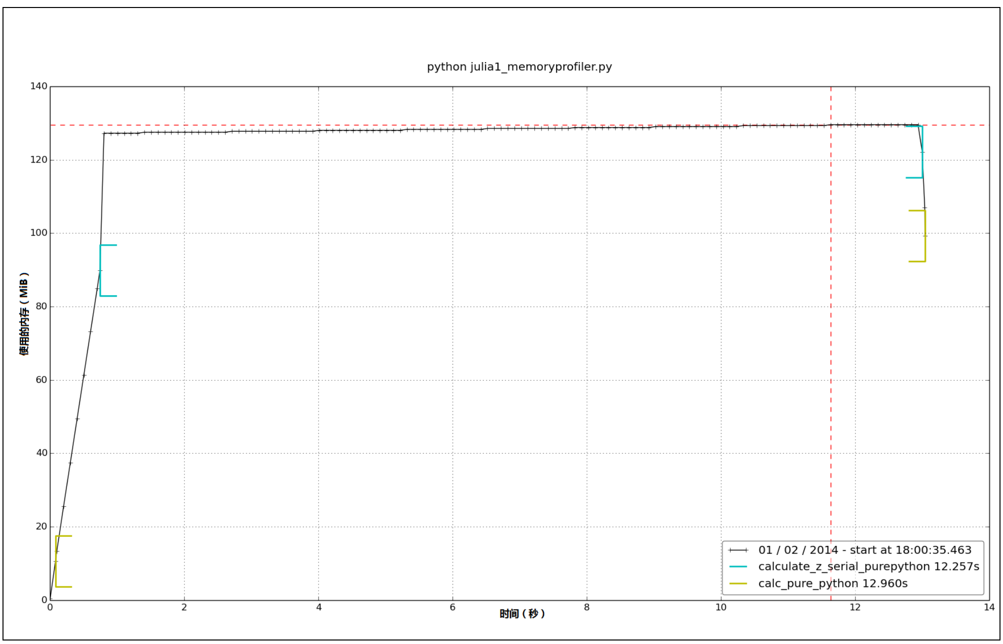


图 2-6　mprof 生成的 memory_profiler 报告

除了在函数层面上观察行为以外，我们还可以使用环境管理器添加标签。例 2-11 的代码用于生成图 2-7。我们可以看到 `create_output_list` 标签：它在 `calculate_z_serial_purepython` 之后出现，并导致进程分配更多 RAM。然后，为了让图更便于理解，我们用 `time.sleep(1)` 暂停 1 秒。

在 `create_range_of_zs` 标签之后，我们看到 RAM 的使用出现了一个猛增；在例 2-11 的代码中，你可以看到这个标签出现在创建 `iterations` 列表时。我们创建列表用的是 `range` 而不是 `xrange`——图显示的很清楚，为了创建一个索引，一个具有 1 000 000 个项目的大列表被实例化。这个方法很没有效率，一旦遇到更大的列表，扩展性也不好（我们的 RAM 会耗尽！）。用于创建这个列表的内存分配操作本身也占用了一点时间，却没有为这个函数带来什么有用的贡献。

备忘

在 Python 3 中 `range` 的行为改变了，它跟 Python 2 的 `xrange` 一样。`xrange` 在 Python 3 中已被淘汰，2to3 转换工具会自动帮你进行这一转换。

例 2-11　用环境管理器给 mprof 图像添加标签

```
@profile
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    with profile.timestamp("create_output_list"):
        output = [0] * len(zs)
    time.sleep(1)
    with profile.timestamp("create_range_of_zs"):
        iterations = range(len(zs))
        with profile.timestamp("calculate_output"):
            for i in iterations:
                n = 0
                z = zs[i]
                c = cs[i]
                while n < maxiter and abs(z) < 2:
                    z = z * z + c
                    n += 1
                output[i] = n
    return output
```

对于占据了图像大部分时间的 calculate_output，我们可以看到 RAM 有一个非常缓慢的线性增长，这是用于分配给内部循环中所有用到的临时数字。使用标签确实可以帮助我们细粒度地了解内存的使用情况。

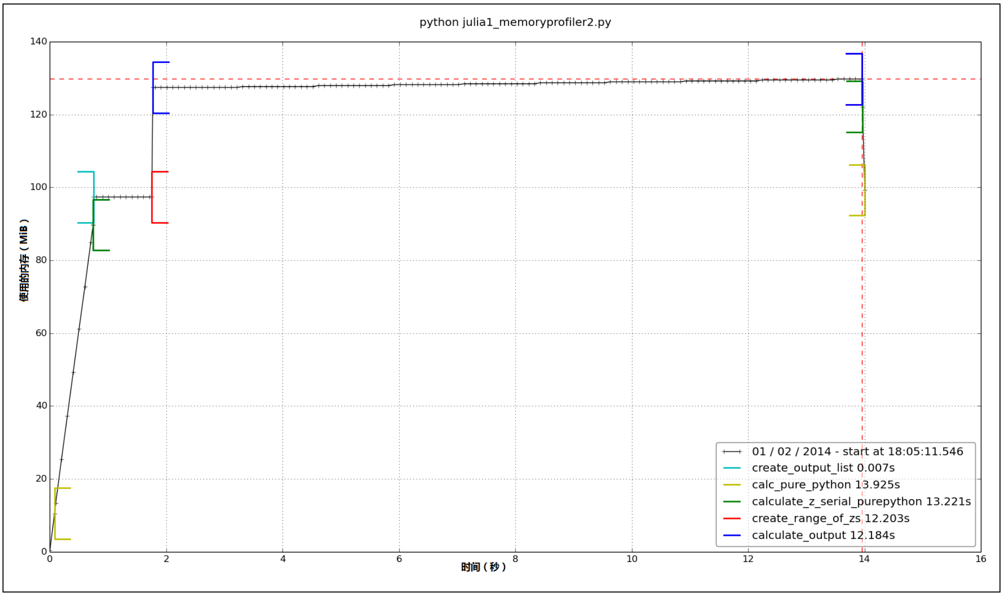


图 2-7　带标签的 mprof 报告

最后，我们可将 `range` 调用替换成 `xrange`。在图 2-8 中，我们可以看到内部循环的 RAM 使用情况相应降低了。

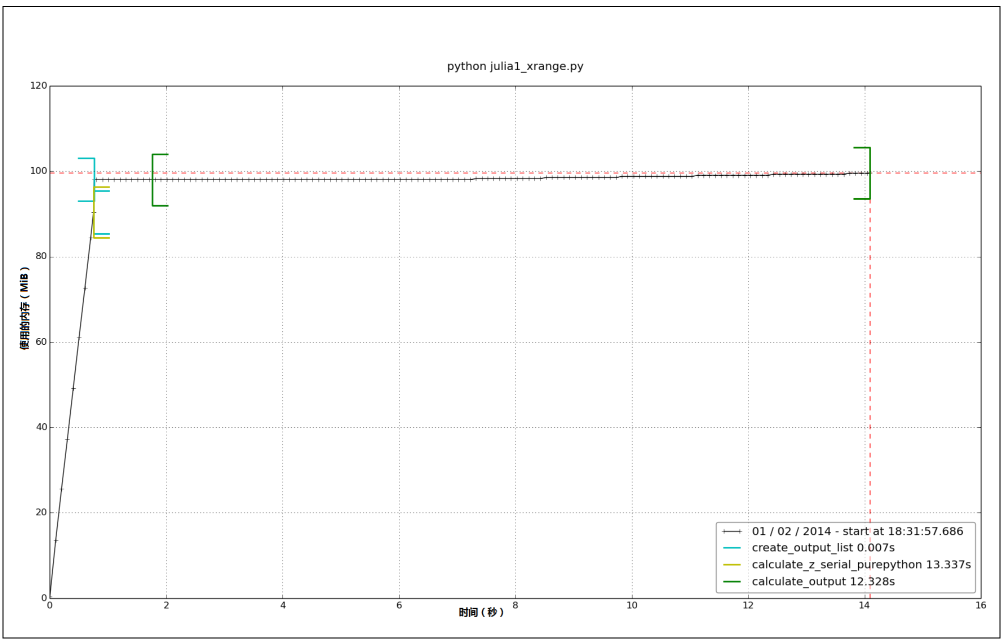


图 2-8　用 xrange 替换 range 之后的效果报告

如果我们想要测量某些语句的 RAM 使用情况，我们可以用 IPython 的`%memit`魔法函数，其工作方式类似于`%timeit`。第 11 章会详细讨论`%memit`的用法以及各种增进 RAM 使用效率的方法。

2.10 用 heapy 调查堆上的对象

Guppy 项目有一个内存堆的调查工具叫作 heapy，可以让你查看 Python 堆中对象的数量以及每个对象的大小。当你需要知道某一时刻有多少对象被使用以及它们是否被垃圾收集时，你尤其需要这种深入解释器内部去了解内存中实际内容的能力。如果你受困于内存泄漏（可能由于你的对象的引用隐藏于一个复杂系统中），那么这个工具能帮你找到问题的关键点。

当你在审查你的代码，看它是否如你预期那样生成对象时，你会发现这个工具非常有用——结果很可能令你吃惊，并为你带来新的优化方向。

为了使用`heapy`，你需要用命令`pip install guppy`安装`guppy`包。

例 2-12 的代码是 Julia 代码的一个略有修改的版本。`calc_pure_python`使用了堆对象`hpy`，我们在三个地方打印堆的内容。

例 2-12　用 heapy 查看代码运行时对象数量的变化

```
def calc_pure_python(draw_output, desired_width, max_iterations):
    ...
    while xcoord < x2:
        x.append(xcoord)
        xcoord += x_step

    from guppy import hpy; hp = hpy()
    print "heapy after creating y and x lists of floats"
    h = hp.heap()
    print h
    print

    zs = []
    cs = []
    for ycoord in y:
        for xcoord in x:
            zs.append(complex(xcoord, ycoord))
            cs.append(complex(c_real, c_imag))

    print "heapy after creating zs and cs using complex numbers"
    h = hp.heap()
    print h
```

```
print

print "Length of x:", len(x)
print "Total elements:", len(zs)
start_time = time.time()
output = calculate_z_serial_purepython(max_iterations, zs, cs)
end_time = time.time()
secs = end_time - start_time
print calculate_z_serial_purepython.func_name + " took", secs, "seconds"

print
print "heapy after calling calculate_z_serial_purepython"
h = hp.heap()
print h
print
```

例 2-13 的输出显示了内存的使用在创建 zs 和 cs 列表后变得有趣：它增长了大约 80MB，因为 2000000 个复数对象消耗了 64000000 字节。这些复数对象占据了目前使用的大多数内存。如果你想要优化这个程序的内存用量，这个结果可用于揭示目前保存的对象数量以及它们总共占用的空间。

例 2-13　heapy 输出显示了我们代码执行时每一个主要阶段的对象总数

```
$ python julia1_guppy.py
heapy after creating y and x lists of floats
Partition of a set of 27293 objects. Total size = 3416032 bytes.
 Index   Count  %    Size    % Cumulative  % Kind (class / dict of class)
     0   10960 40 1050376   31    1050376  31 str
     1    5768 21  465016   14    1515392  44 tuple
     2     199  1  210856    6    1726248  51 dict of type
     3      72  0  206784    6    1933032  57 dict of module
     4    1592  6  203776    6    2136808  63 types.CodeType
     5     313  1  201304    6    2338112  68 dict (no owner)
     6    1557  6  186840    5    2524952  74 function
     7     199  1  177008    5    2701960  79 type
     8     124  0  135328    4    2837288  83 dict of class
     9    1045  4   83600    2    2920888  86 __builtin__.wrapper_descriptor
<91 more rows. Type e.g. '_.more' to view.>

heapy after creating zs and cs using complex numbers
Partition of a set of 2027301 objects. Total size = 83671256 bytes.
 Index   Count  %     Size    % Cumulative  % Kind (class / dict of class)
     0 2000000  99  6400000  76   64000000  76 complex
     1     185   0 16295368  19   80295368  96 list
     2   10962   1  1050504   1   81345872  97 str
     3    5767   0   464952   1   81810824  98 tuple
     4     199   0   210856   0   82021680  98 dict of type
     5      72   0   206784   0   82228464  98 dict of module
```

```
     6   1592   0   203776   0  82432240  99 types.CodeType
     7    319   0   202984   0  82635224  99 dict (no owner)
     8   1556   0   186720   0  82821944  99 function
     9    199   0   177008   0  82998952  99 type
<92 more rows. Type e.g. '_.more' to view.>

Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 13.2436609268 seconds

heapy after calling calculate_z_serial_purepython
Partition of a set of 2127696 objects. Total size = 94207376 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0 2000000  94 64000000  68  64000000  68 complex
     1    186   0 24421904  26  88421904  94 list
     2 100965   5  2423160   3  90845064  96 int
     3  10962   1  1050504   1  91895568  98 str
     4   5767   0   464952   0  92360520  98 tuple
     5    199   0   210856   0  92571376  98 dict of type
     6     72   0   206784   0  92778160  98 dict of module
     7   1592   0   203776   0  92981936  99 types.CodeType
     8    319   0   202984   0  93184920  99 dict (no owner)
     9   1556   0   186720   0  93371640  99 function
<92 more rows. Type e.g. '_.more' to view.>
```

第三段显示了在计算完 Julia 集合后，我们占用了 94MB 的内存。除了之前的复数，我们现在还保存了大量的整数，列表中的项目也变多了。

`hpy.setrelheap()`可以用来创建一个内存断点，当后续调用 `hpy.heap()`时就会产生一个跟这个断点的差额。这样你就可以略过断点前由 Python 内部操作导致的内存分配。

2.11 用 dowser 实时画出变量的实例

Robert Brewer 的 dowser 可以在代码运行时钩入名字空间并通过 CherryPy 接口在一个 Web 服务器上提供一个实时的变量实例图。每个被追踪对象都有一个走势图，让你可以看到某个对象的数量是否在增长。这在分析长期运行的进程时很有用。

如果你有一个长期运行的进程且你预计程序的不同操作会带来不同的内存变化（比如你可能想对一台 web 服务器上传一些数据或跑一些复杂的查询），那么你可以实时确认这些变化，见图 2-9。

要使用它，我们需要在 Julia 代码中加入辅助函数（例 2-14）用来启动 CherryPy 服务器。

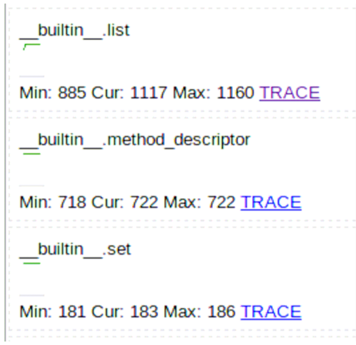


图 2-9　dowser 通过 CherryPy 显示的三张走势图

例 2-14　在应用中启动 dowser 的辅助函数

```
def launch_memory_usage_server(port=8080):
    import cherrypy
    import dowser

    cherrypy.tree.mount(dowser.Root())
    cherrypy.config.update({
        'environment': 'embedded',
        'server.socket_port': port
    })

    cherrypy.engine.start()
```

在开始计算之前，我们需要首先启动 CherryPy 服务器，如例 2-15 所示。完成计算后，我们可以调用 `time.sleep` 维持控制台打开——这会让 CheeryPy 进程保持运行，让我们可以继续审查名字空间的状态。

例 2-15　在正确的时机启动 dowser，这会启动一个 Web 服务器

```
...
        for xcoord in x:
            zs.append(complex(xcoord, ycoord))
            cs.append(complex(c_real, c_imag))

    launch_memory_usage_server()

...
    output = calculate_z_serial_purepython(max_iterations, zs, cs)
...
    print "now waiting..."
while True:
    time.sleep(1)
```

点击图 2-9 中的 TRACE 链接，我们就可以看到每个 list 对象的内容（图 2-10）。我们还可以继续深入每个 list 内部——这就像在 IDE 中使用一个交互调试器一样，

但是你可以在一台服务器上进行而*不需要*一个 IDE。

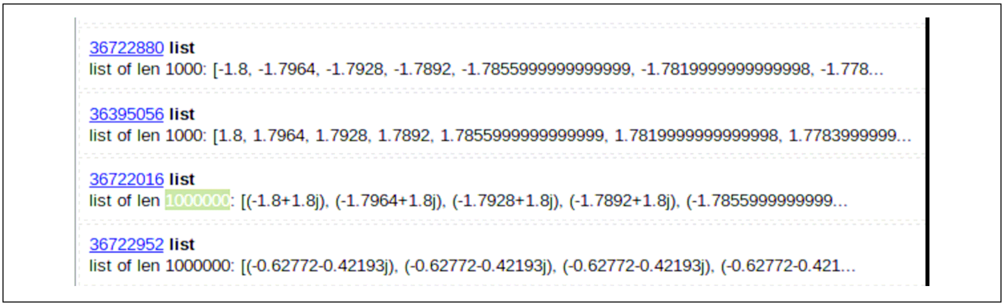


图 2-10　dowser 显示某个列表中有 1 000 000 个项目

备忘

我们当然更希望在可控的情况下直接分析代码块。但是有时这不一定做得到，或者有时你只是想要简单地分析一下情况。查看一个正在运行的进程的内部细节会是一种折中的方案，不需要进行太多的工作就可以提供足够的证据。

2.12　用 dis 模块检查 CPython 字节码

到目前为止我们已经展示了很多测量 Python 代码开销的方法（包括 CPU 和 RAM 的开销）。不过，我们还没有看到在底层虚拟机的字节码层面发生的事情。了解“台面下”发生的事情有助于在脑海中对运行慢的函数建立一个模型，并能帮助你编译你的代码。所以现在让我们来看一些字节码。

`dis` 模块让我们能够看到基于栈的 CPython 虚拟机中运行的字节码。在你的 Python 代码运行的时候，了解虚拟机中发生了什么可以帮助你了解为什么某些编码风格会比其他的更快。同时还能帮助你使用 Cython 这样的工具，它跳出了 Python 的范畴，能够生成 C 代码。

`dis` 模块是内建的。你可以传给它一段代码或者一个模块，它会打印出分解的字节码。在例 2-16 中我们分解了函数的外层循环。

问题

你应该试着分解一个你自己的函数并将每一个分解的代码和分解的输出匹配起来。你能匹配下面的 dis 输出和原始函数吗？

例 2-16　使用内建的 dis 模块来了解运行代码的虚拟机

```
In [1]: import dis
In [2]: import julia1_nopil

In [3]: dis.dis(julia1_nopil.calculate_z_serial_purepython)
 11           0 LOAD_CONST               1 (0)
              3 BUILD_LIST               1
              6 LOAD_GLOBAL              0 (len)
              9 LOAD_FAST                1 (zs)
             12 CALL_FUNCTION            1
             15 BINARY_MULTIPLY
             16 STORE_FAST               3 (output)

 12          19 SETUP_LOOP             123 (to 145)
             22 LOAD_GLOBAL              1 (range)
             25 LOAD_GLOBAL              0 (len)
             28 LOAD_FAST                1 (zs)
             31 CALL_FUNCTION            1
             34 CALL_FUNCTION            1
             37 GET_ITER
          >> 38 FOR_ITER               103 (to 144)
             41 STORE_FAST               4 (i)

 13          44 LOAD_CONST               1 (0)
             47 STORE_FAST               5 (n)
# ...
# We'll snip the rest of the inner loop for brevity!
# ...

 19     >> 131 LOAD_FAST                5 (n)
           134 LOAD_FAST                3 (output)
           137 LOAD_FAST                4 (i)
           140 STORE_SUBSCR
           141 JUMP_ABSOLUTE           38
        >> 144 POP_BLOCK

 20     >> 145 LOAD_FAST                3 (output)
           148 RETURN_VALUE
```

这个输出非常的直白简明。第一列包含了原始文件的行数。第二列包含了一些>>标志，它们是指向其他代码的跳转点。第三列是操作的地址和操作名。第四列包含了操作的参数。第五列的标记可用来帮助对照字节码和原始 Python 的参数。

对照字节码和例 2-3 中的 Python 代码。字节码首先将常数 0 放到栈上，然后创建了一个具有一个项目的列表。接下来，它搜索了名字空间来查询 `len` 函数，将它放到栈上，再次搜索名字空间找到 `zs`，放到栈上。在第 12 行，它从栈上调用 `len` 函数，且弹出了栈上的 `zs` 作为参数；然后对最后两个参数调用二进制乘法（`zs`

的长度和那个单项目列表）将结果保存在 output 中。这就是我们那个 Python 函数的第一行干的事情。你可以继续查看下一段字节码来了解 Python 代码第二行的行为（外层 for 循环）。

问题

跳转点（>>）匹配 JUMP_ABSOLUTE 以及 POP_JUMP_IF_FALSE 等指令。过一遍你自己的函数分解结果并对照跳转点和跳转指令。

介绍完字节码，我们现在要问：要完成同样的任务，显式编写的函数和使用内建函数在字节码和时间开销上的对比是什么。

不同的方法，不同的复杂度

> 应该有一种——而且最好只有唯一的一种——明显的方式去完成它。虽然这种方式可能一开始并不明显，除非你是荷兰人……
>
> ——Tim Peters
>
> Python 之禅

通过 Python 你有无数种方式表达你的意思。一般来说最优的那个选择十分明显，但是如果你的经验主要来自一个老版本的 Python 或另一门编程语言，那么在你的脑海里可能就是另外的选择。某些表达的方式可能就比别的要慢。

对于你大多数的代码，你可能更关心可读性而不是速度，这能让你的团队更有效地写代码，而不是被高效而难懂的代码所迷惑。但是，有些时候你会更追求性能（且不牺牲可读性），那么你需要的可能是一些速度测试。

看看例 2-17 的两段代码。它们都做了相同的工作，但是第一个会产生大量额外的 Python 字节码，带来更大的开销。

例 2-17　一个单纯的和一个高效的手段解决同一个求和问题

```
def fn_expressive(upper = 1000000):
    total = 0
    for n in xrange(upper):
        total += n
    return total

def fn_terse(upper = 1000000):
    return sum(xrange(upper))

print "Functions return the same result:", fn_expressive() == fn_terse()
```

```
Functions return the same result:
True
```

两个函数都对一批整数求和。一个简单的经验法则（但是你一*定*要进行性能分析！）是字节码越多执行的速度越慢。内建函数使用了更少的字节码行数来完成同样的工作。我们在例 2-18 中使用 IPython 的`%timeit`魔法函数测量它们的最快运行时间。

例 2-18　用%timeit 验证我们的假设：内建函数应该比自己写函数要快（译注：原著中未给出 fn_terse()的结果）

```
%timeit fn_expressive()

10 loops, best of 3: 42 ms per loop
100 loops, best of 3: 12.3 ms per loop

%timeit fn_terse()
```

如果我们用 `dis` 模块调查每个函数的字节码，如例 2-19 所示，我们能看到虚拟机用了 17 行来执行更有表现力的函数，而仅用了 6 行来执行非常可读但更简洁的第二个函数。

例 2-19　用 dis 查看两个函数的字节码指令行数

```
import dis
print fn_expressive.func_name
dis.dis(fn_expressive)

fn_expressive
  2           0 LOAD_CONST               1 (0)
              3 STORE_FAST               1 (total)

  3           6 SETUP_LOOP              30 (to 39)
              9 LOAD_GLOBAL              0 (xrange)
             12 LOAD_FAST                0 (upper)
             15 CALL_FUNCTION            1
             18 GET_ITER
        >>   19 FOR_ITER                16 (to 38)
             22 STORE_FAST               2 (n)

  4          25 LOAD_FAST                1 (total)
             28 LOAD_FAST                2 (n)
             31 INPLACE_ADD
             32 STORE_FAST               1 (total)
             35 JUMP_ABSOLUTE           19
        >>   38 POP_BLOCK

  5     >>   39 LOAD_FAST                1 (total)
             42 RETURN_VALUE
```

```
print fn_terse.func_name
dis.dis(fn_terse)

fn_terse
  8           0 LOAD_GLOBAL              0 (sum)
              3 LOAD_GLOBAL              1 (xrange)
              6 LOAD_FAST                0 (upper)
              9 CALL_FUNCTION            1
             12 CALL_FUNCTION            1
             15 RETURN_VALUE
```

两段代码的区别很明显。在 `fn_expressive()`内部，我们维护了两个本地变量并用 `for` 循环遍历了一个列表。`for` 循环会在每次循环时检查 StopIteration 异常是否被引发。每次迭代都会调用 `total.__add__`函数，这个函数会检查第二个变量（`n`）的类型。所有这些检查都会带来一些开销。

在 `fn_terse()`内部，我们调用了一个用 C 编写的优化的列表操作函数，它知道如何生成最后的结果而无须创建中间的 Python 对象。这样会快很多，即使每次迭代仍然必须检查被求和对象的类型（我们会在第 4 章看到一些将类型固定的方法，这样就不需要在每次迭代时都检查它）。

之前提过，你一*定要*对你的代码进行性能分析——但如果你只依靠性能分析来试错，那你必然会在某些时候写出较慢的代码。学习 Python 是否已经存在一个内建的更短且依然可读的方式来解决你的问题是绝对值得的。如果已经存在，那么它可能更容易被另一个开发人员理解且*可能*运行得会更快。

2.13　在优化期间进行单元测试保持代码的正确性

如果你不对你的代码进行单元测试，那么从长远来看你可能正在损害你的生产力。Ian（脸红）十分尴尬地提到有一次他花了一整天的时间优化他的代码，因为嫌麻烦所以他禁用了单元测试，最后却发现那个显著的速度提升只是因为他破坏了需要优化的那段算法。这样的错误你一次都不要犯。

除了单元测试，你还应该坚定地考虑使用 `coverage.py`。它会检查有哪些代码行被你的测试所覆盖并找出那些没有被覆盖的代码。这可以让你迅速知道你是否测试了你想要优化的代码，那么在优化过程中可能潜伏的任何错误都会被迅速抓出来。

No-op 的@profile 修饰器

如果你的代码使用了 `line_profiler` 或者 `memory_profiler` 的`@profile` 修

饰器，那么你的单元测试会引发一个 NameError 异常并失败。原因是单元测试框架不会将@profile 修饰器注入本地名字空间。no-op 修饰器可以在这种时候解决问题。在你测试时把它加入你的代码块，并在你结束测试后移除它是在方便不过的事情了。

使用 no-op 修饰器，你可以运行你的测试而不需要修改你的代码。这意味着你可以在每次优化之后都运行你的测试，你将永远不会倒在一个出问题的优化步骤上。

如例 2-20 所示，假设我们有一个 ex.py 模块，它有一个测试用例（基于 nosetests 框架）和一个函数，这个函数我们正在用 line_profiler 或者 memory_profiler 进行性能分析。

例 2-20　一个简单的函数和一个测试用例需要用到@profile

```
# ex.py
import unittest

@profile
def some_fn(nbr):
    return nbr * 2


class TestCase(unittest.TestCase):
    def test(self):
        result = some_fn(2)
        self.assertEquals(result, 4)
```

如果我们运行 nosetests 测试我们的代码就会得到一个 NameError：

```
$ nosetests ex.py
E
======================================================================
ERROR: Failure: NameError (name 'profile' is not defined)
...
NameError: name 'profile' is not defined
Ran 1 test in 0.001s

FAILED (errors=1)
```

解决方法是在 ex.py 开头添加一个 no-op 修饰器（你可以在完成性能分析之后移除它）。如果在名字空间中寻找不到@profile 修饰器（因为没有使用 line_profiler 或者 memory_profiler），那么我们写的 no-op 版本的修饰器就会被加入名字空间。如果 line_profiler 或者 memory_profiler 已经将新的函数加入名字空间，那么我们 no-op 版本的修饰器就会被忽略。

对于 line_profiler，我们可以加入例 2-21 的代码。

例 2-21　在单元测试时在名字空间中加入针对 line_profiler 的 no-op@profile 修饰器

```
# line_profiler
if '__builtin__' not in dir() or not hasattr(__builtin__, 'profile'):
    def profile(func):
        def inner(*args, **kwargs):
            return func(*args, **kwargs)
        return inner
```

`__builtin__`检查是针对 `nosetests` 的，`hasattr` 则用来检查`@profile` 修饰器是否已经被加入名字空间。现在可以在我们的代码上成功运行 `nosetests` 了：

```
$ kernprof.py -v -l ex.py
Line #      Hits         Time    Per %%HTMLit    % Time    Line Contents
==============================================================
    11                                                  @profile
    12                                                  def some_fn(nbr):
    13         1            3       3.0    100.0      return nbr * 2


$ nosetests ex.py
.
Ran 1 test in 0.000s
```

对于 `memory_profiler`，我们使用例 2-22 的代码。

例 2-22　在单元测试时在名字空间中加入针对 memory_profiler 的 no-op@profile 修饰器

```
# memory_profiler
if 'profile' not in dir():
    def profile(func):
        def inner(*args, **kwargs):
            return func(*args, **kwargs)
        return inner
```

期望产生的输出如下：

```
python -m memory_profiler ex.py
...
Line #    Mem usage    Increment   Line Contents
================================================
    11   10.809 MiB    0.000 MiB   @profile
    12                             def some_fn(nbr):
    13   10.809 MiB    0.000 MiB       return nbr * 2


$ nosetests ex.py
.
Ran 1 test in 0.000
```

不使用这些修饰器可以节省你几分钟，但是一旦你在一个破坏你代码的错误优化上失去了好几个小时，你就会想要把这个加入你的工作流程了。

2.14 确保性能分析成功的策略

性能分析需要一些时间和精力。如果你把需要测试的代码段跟你代码的主体分离，你会有一个更好的机会去了解你的代码。然后你可以用单元测试来保证正确性，你还可以传入精心编造的真实数据来测试算法的有效性。

记得关闭任何基于 BIOS 的加速器，因为它们只会混淆你的结果。Ian 的笔记本电脑使用的 Intel TurboBoost 功能可以在温度足够低的时候将 CPU 暂时加至极速。这意味着低温时运行同一段代码的速度可能比高温时要快。你的操作系统也许还控制了时钟的速度——使用电池电源的笔记本可能比插了主电源时更积极地控制 CPU 的速度。为了建立一个更加稳定的测试配置，我们：

- 在 BIOS 上禁用了 TurboBoost。
- 禁用了操作系统改写 SpeedStep（如果你有权限，你可以在你的 BIOS 中找到它）的能力。
- 只使用主电源（从不使用电池电源）。
- 运行实验时禁用后台工具如备份和 Dropbox。
- 多次运行实验来获得一个稳定的测量结果。
- 如果可能，降至 run level 1（UNIX），确保没有其他任务运行。
- 重启并重跑实验来二次验证结果。

试着假设你代码的行为并用性能分析的结果来证实（或证伪）你的假设。你的选择不会改变（因为你的决定只能基于性能分析后的结果），但是你对代码的直觉了解会提升，而这会在今后的项目中带来好处，因为你会变得更能做出高效的决定。当然，你依然需要性能分析来验证这些高效的决定。

不要克扣准备工作。如果你在测试一段深入大型项目的代码前不先将代码分离，你很有可能会因为一些副作用而让你的努力偏离正轨。当你进行细粒度的改动时，对大型项目进行单元测试往往会更困难，而这又会更进一步妨碍你的努力。副作用可能包括其他线程或进程影响了 CPU 和内存的使用以及网络和磁盘的活动，这些都会歪曲你的结果。

对于 Web 服务器，推荐 dowser 和 dozer；你可以用它们来将名字空间中的对象行为实时可视化。如果可能，一定要将你想测试的代码从 Web 应用的主体上分离出来，这会让性能分析方便太多。

确保你的单元测试覆盖了所有你想要分析的代码路径。任何你没有测过的东西都有可能带来细微的错误拖慢你的进度。使用 coverage.py 来确认你的测试覆盖了所有的代码路径。

对一个生成很多数字输出的复杂代码段进行单元测试可能会很困难。不要害怕将结果输出到一个文本文件来运行 diff 或者使用一个 pickled 对象。对于数字优化的问题，Ian 喜欢创建一个包含了大量浮点数的长文本文件并使用 diff——细小的取整问题会立刻显现，哪怕它们在输出中很罕见。

如果你的代码容易受到数字取整问题的影响，那么你最好有一个大的输出可以用来进行前后对比。取整错误的一个原因是 CPU 寄存器和主存之间的浮点精度不同。你的代码在不同的代码路径上运行可能导致细微的取整错误并在之后给你带来困扰——所以最好在它们刚发生的时候就尽早意识到这点。

显然，在性能分析和优化时使用源代码控制工具是很有意义的。创建新的代码分支代价很低，而且它能让你保持头脑清醒。

2.15 小结

看过各种性能分析技术以后，你应该已经有了所有必需的工具来验证你的代码中的 CPU 和 RAM 瓶颈。接下来我们要去看看 Python 是如何实现最常用的容器的，这样你就能明智地决定使用哪种容器来存放大数据的集合。

第 3 章

列表和元组

读完本章之后你将能够回答下列问题

- 列表和元组各自适用于什么情况?
- 查询列表/元组的复杂度是什么?
- 该复杂度是如何计算出来的?
- 列表和元组的区别是什么?
- 向列表添加新项目是如何实现的?
- 我应该在什么情况下使用列表和元组?

写高性能程序最重要的事情是了解你的数据结构所能够提供的性能保证。事实上，高性能编程的很大一部分是了解你查询数据的方式，并选择一个能够迅速响应这个查询的数据结构。本章我们将谈论那些列表和元组能迅速响应的查询，以及它们是如何响应的。

列表和元组之类的数据结构被称为*数组*。一个数组是数据在某种内在次序下的扁平列表。这一*先验*次序十分重要：知道了数据在数组中的确定位置，我们就能以`O(1)`的复杂度得到它！另外，数组可以有多种实现方式。下面是列表和元组的另一个区别：列表是动态的数组，而元组则是静态的数组。

让我们回顾一下之前的定义。一台计算机的系统内存可以被看作是一系列编了号的桶，每个桶可以存放一个数字。这些数字可以被用来代表任何我们关心的变量（整数、浮点数、字符串，或其他数据结构），因为它们只是引用了数据被

保存在内存中的位置[①]。

当我们想要创建一个数组（也就是一个列表或元组）时，我们首先必须分配一块系统内存（其每一段都将被当成是一个整型大小的指向实际数据的指针）。这需要进入内核，操作系统的子进程，去申请使用 *N* 个*连续*的桶。图 3-1 的例子显示了一个大小为 6 的列表的系统内存布局。注意在 Python 中列表还记录了它们的大小，所以在分配的 6 个块中，仅可使用 5 个——第一个元素是列表的长度。

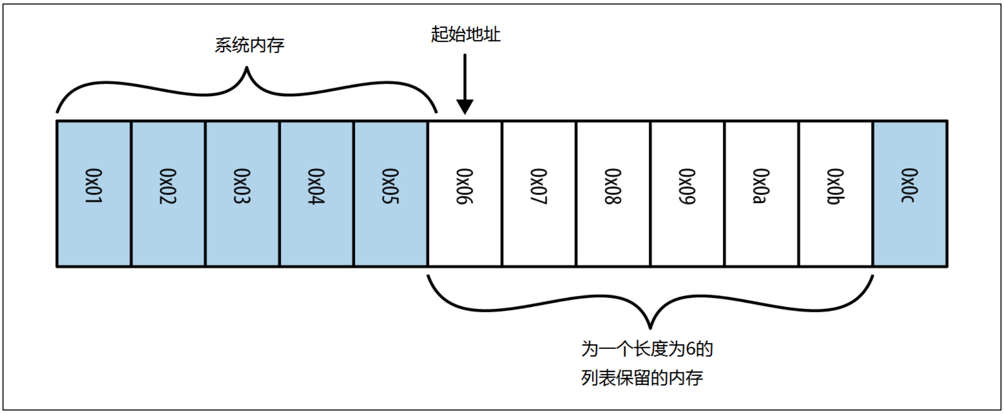


图 3-1　一个长度为 6 的数组的系统内存布局示例

为了能在我们的列表中查询任意指定元素，我们只需要知道我们想要的是哪个元素以及放数据的起始桶是哪个即可。因为所有的数据都占据同样大小的空间（一个“桶”，或者更确切地说，一个整型大小的指向实际数据的指针），我们不需要知道任何关于被存储数据的类型的信息就能够进行计算。



问题

如果你知道你的 N 元素列表从哪里开始，那么你如何查找列表中任意的元素？

假设我们需要获取数组的第一个元素，只需要去第一个桶，`M+1`，并读出其中的值。（译注：原文这里是 `M`，但是根据上下文应为 `M+1`。）另一方面，如果我们需要数组的第五个元素，可以去位于 `M+5` 的桶并读取其内容。总而言之，如果我们想要获取数组的第 `i` 个元素，就去桶 `M+i`。也就是说，只要我们的数据保存在连续的桶里且知道数据的顺序，就能一步（`O(1)`）定位到数据所在的桶，无论我们的数组有多大（见例 3-1）。

① 在 64 位计算机里，12KB 内存可以给你 725 个桶，而 52GB 内存可以给你 3 250 000 000 个桶！

例 3-1　对不同大小的列表进行查询的时间

```
>>> %%timeit l = range(10)
        ...: l[5]
        ...:
10000000 loops, best of 3: 75.5 ns per loop
>>>
>>> %%timeit l = range(10000000)
        ...: l[100000]
        ...:
10000000 loops, best of 3: 76.3 ns per loop
```

如果我们拿到的是一个未知次序的数组，又该如何获取某个元素？如果是次序已知的数组，我们可以直接查询指定的值。而在次序未知的情况下，我们必须进行搜索操作。解决这个问题最基本的方法叫“线性搜索”，我们遍历数组中的每一个元素并检查其值是否为我们想要的，见例 3-2。

例 3-2　对列表进行线性搜索

```
def linear_search(needle, array):
    for i, item in enumerate(array):
        if item == needle:
            return i
    return -1
```

这个算法的最差情况性能是 `O(n)`。最差情况发生在当我们搜索的东西不存在于数组中时。为了知道我们搜索的元素不存在于数组中，我们必须对所有的元素进行检查。最终我们走到了最后的 `return -1` 语句。事实上，这个算法正是 `list.index()` 使用的算法。

唯一提升速度的方法是了解数据如何存放在内存中，或者说了解存放我们数据的桶的组织方式。比如，散列表，一个用于实现字典和集合的基本数据结构，通过丢弃数据的原始次序并指定另外一种次序，或者更确切地说，另外一种数据组织方式，能够以 `O(1)` 的复杂度解决这个问题。又比如说，如果你的数据是排过序的，每一个元素都大于（或小于）位于其左边（或右边）的相邻元素，那么一个特化的搜索算法可以将你的搜索时间降至 `O(log n)`。我们之前的常数时间的查找不需要排序这一步骤，然而在某些时候，先排序后搜索是最佳的选择（特别是因为这可以让搜索算法更为灵活且允许你创出新的搜索方式）。

问题

给定下列数据，写一个算法找到值 61 的索引：

```
[9, 18, 18, 19, 29, 42, 56, 61, 88, 95]
```

已知数据经过排序，你如何可以做得更快？

提示：如果你将数组对半分，你知道左半边所有的值都小于右半边最小的元素。你可以利用这点！

3.1　一个更有效的搜索

前文中已经暗示了，如果我们先对我们的数据进行排序，使得所有位于某个元素左边的其他元素都小于（或大于）它，那么就可以获得更好的搜索性能。对象的比较是通过对象的魔法函数`__eq__`和`__lt__`进行的，用户对象可以自定义这两个函数。

备忘

没有`__eq__`和`__lt__`方法，一个用户对象就只能跟同类型的对象比较且比的是对象实例在内存中的地址。

高效搜索必需的两大要素是排序算法和搜索算法。Python 列表有一个内建的排序算法使用了 Tim 排序。Tim 排序可以在最佳情况下以`O(n)`（最差情况下则是`O(n log n)`）的复杂度排序。它运用了多种排序算法。对于给定的数据，它使用探索法猜测哪个算法的性能最优（更确切地说，它混用了插入排序和合并排序算法）来达到这样的性能。

一旦一个列表被排序，我们就可以用二分搜索找到我们的目标（例 3-3），其平均情况复杂度是`O(log n)`。它首先查询位于列表中点的值并和目标值比较。如果中点值小于目标值，我们就继续考察右半边列表，我们不断将列表二分，直至找到目标值或发现该值不存在于列表。结果就是我们不需要像线性搜索那样读取列表中所有的元素，而仅仅读取了一个子集。

例 3-3　对已排序列表的高效搜索——二分搜索

```
def binary_search(needle, haystack):
    imin, imax = 0, len(haystack)
    while True:
        if imin >= imax:
            return -1
        midpoint = (imin + imax) // 2
        if haystack[midpoint] > needle:
            imax = midpoint
        elif haystack[midpoint] < needle:
            imin = midpoint+1
        else:
            return midpoint
```

这个方法使得我们无须求助重量级的字典解决方案就能够查找列表中的元素。特别是当列表本身就已经排过序的情况下，用二分搜索进行对象查找（复杂度 O(log n)）比将列表转换成字典然后进行查询要更高效（虽然字典查找复杂度是 O(1)，字典转换复杂度却是 O(n)。而且字典要求没有重复的键，你可能不希望受到这样的限制）。

另外，使用 bisect 模块可以进一步简化这一流程，它提供了一个简便的函数让你可以在保持排序的同时往列表中添加元素，以及一个高度优化过的二分搜索算法函数来查找元素。它提供的函数可以将新元素直接插入正确的排序位。在列表始终排序的情况下，我们可以轻松找到需要的元素（示例代码可以在 bisect 模块的文档中找到）。另外，我们可以非常迅速地用 bisect 找到跟我们的目标值最接近的元素（例 3-4）。这个功能对于比较两个相似但不完全一样的数据集来说极其有用。

例 3-4 用 bisect 模块在列表中查询最接近目标的值

```
import bisect
import random

def find_closest(haystack, needle):
    # bisect.bisect_left will return the first value in the haystack
    # that is greater than the needle
    i = bisect.bisect_left(haystack, needle)
    if i == len(haystack):
        return i - 1
    elif haystack[i] == needle:
        return i
    elif i > 0:
         j = i - 1
        # since we know the value is larger than needle (and vice versa for the
        # value at j), we don't need to use absolute values here
       if haystack[i] - needle > needle - haystack[j]:
          return j
    return i

important_numbers = []
for i in xrange(10):
    new_number = random.randint(0, 1000)
    bisect.insort(important_numbers, new_number)

# important_numbers will already be in order because we inserted new elements
# with bisect.insort
print important_numbers

closest_index = find_closest(important_numbers, -250)
print "Closest value to -250: ", important_numbers[closest_index]
```

```
closest_index = find_closest(important_numbers, 500)
print "Closest value to 500: ", important_numbers[closest_index]

closest_index = find_closest(important_numbers, 1100)
print "Closest value to 1100: ", important_numbers[closest_index]
```

总的来说，本节涉及了编写高效代码的基本原则：选择正确的数据结构并坚持使用它！虽然对于某个特定操作来说也许还存在更高效的数据结构，但是在这些数据结构之间进行转换的代价可能会抵消效率上的增益。

3.2 列表和元组

如果列表和元组都使用了相同的数据结构，那么两者之间还有什么区别？主要区别总结如下：

1．列表是动态数组，它们可变且可以重设长度（改变其内部元素的个数）。

2．元组是静态数组，它们不可变，且其内部数据一旦创建便无法改变。

3．元组缓存于 Python 运行时环境，这意味着我们每次使用元组时无须访问内核去分配内存。

这些区别揭示了两者在设计哲学上的不同：元组用于描述一个不会改变的事物的多个属性，而列表可被用于保存多个互相独立对象的数据集合。比如，保存一个电话号码适合用元组：它们不会改变，如果改变则意味着他们代表了一个新的对象，也就是另一个电话号码。同样，保存一个多项式的系数适合用元组，因为不同的系数代表了不同的多项式。另一方面，保存当前正在阅读本书的人的名字更适合用列表：虽然数据的内容和大小时刻在发生变化，但始终表示同一个概念。

值得提醒的是，列表和元组都可以接受混合类型。我们会看到，这会带来一些额外的开销并减少一些可能的优化。如果我们强制要求所有的数据都是同一个类型，那么就可以避免这些开销。我们将在第 6 章讨论如何通过使用 `numpy` 降低内存和计算的开销。另外，对于非数字的数据，还有一些其他模块，如 `blist` 和 `array` 也能够减少这些开销。这暗示了我们将在后续章节介绍的高性能编程的一个主要要点：通用代码会比为某个特定问题设计的代码慢很多。

另外，跟列表可以改变大小及内容不同，元组的不可改变性使其成为了一个非常轻量级的数据结构。这意味着存储它们不需要很多的内存开销，而且对它的操作也非常的直观。我们将会看到列表的可变性的代价在于存储它们需要额外的内存以及使用它们需要额外的计算。

问题

对于下面的示例数据集，你会选择元组还是列表？为什么？

1. 前 20 个质数。
2. 编程语言的名字。
3. 一个人的年龄、体重、身高。
4. 一个人的生日和出生地。
5. 某次台球游戏的结果。
6. 一系列台球游戏的结果。

答案:

1. 元组，因为数据是静态的且不会改变。
2. 列表，因为数据集会不停增长。
3. 列表，因为这些值会被更新。
4. 元组，因为这些信息是静态的且不会改变。
5. 元组，因为数据是静态的。
6. 列表，因为会有更多游戏进行（事实上，我们可以使用一个元组的列表。因为单个游戏的数据不会改变，但是随着游戏次数的上升，我们会需要增加列表的长度）。

3.2.1 动态数组：列表

一旦我们创建了列表，我们就可以根据需要随意改变其内容：

```
>>> numbers = [5, 8, 1, 3, 2, 6]
>>> numbers[2] = 2*numbers[0] # ❶
>>> numbers
[5, 8, 10, 3, 2, 6]
```

❶ 如前所述，这个操作是 `O(1)`，因为我们可以立即找到第 0 个和第 2 个数据保存的位置。

另外，我们可以给列表添加新的数据来增加其大小：

```
>>> len(numbers)
6
>>> numbers.append(42)
>>> numbers
[5, 8, 10, 3, 2, 6, 42]
>>> len(numbers)
7
```

这是因为动态数组支持 `resize` 操作，可以增加数组的容量。当一个大小为 `N` 的列表第一次需要添加数据时，Python 会创建一个新的列表，足够存放原来的 `N` 个元素以及额外需要添加的元素。不过，实际分配的并不是 `N+1` 个元素，而是 `M` 个，`M > N`，这是为了给未来的添加预留空间。然后旧列表的数据被复制到新列表中，旧列表则被销毁。从设计理念上来说，第一次的添加可能会是后续多次添加的开始，通过预留空间的做法，我们就可以减少这一分配空间的操作的次数以及内存复制的次数。这一点非常重要，因为内存复制可能非常昂贵，特别是当列表大小开始增长以后。图 3-2 显示了在 Python 2.7 中这一超额分配的做法。分配空间的公式见例 3-5。

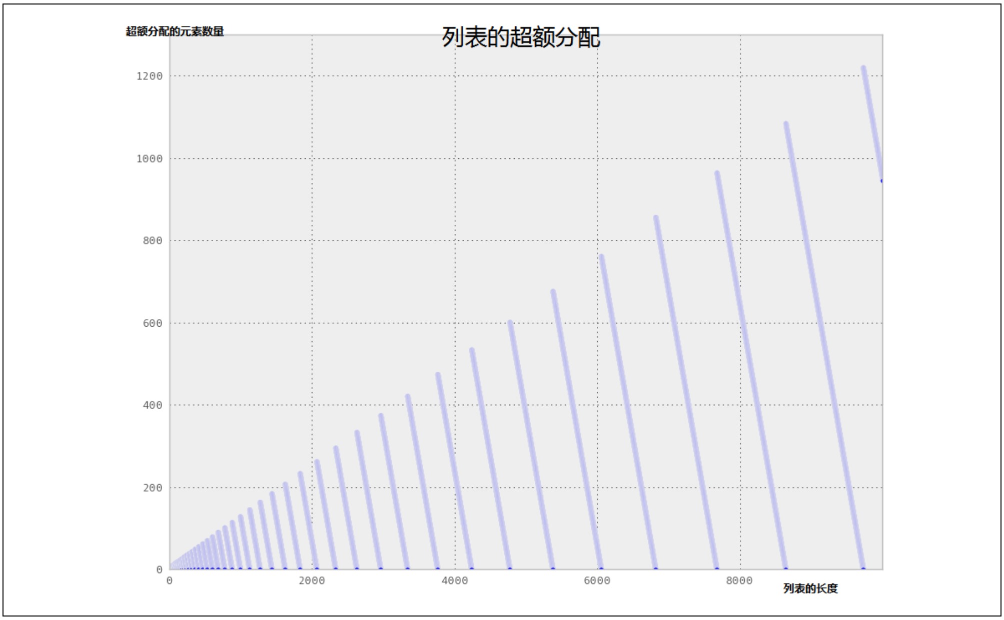


图 3-2　图中显示了对于一个特定大小的列表会分配多少个额外的元素

例 3-5　Python 2.7 的列表空间分配公式

```
M = (N >> 3) + (N < 9 ? 3 : 6)
```

N	0	1-4	5-8	9-16	17-25	26-35	36-46	…	991-1120
M	0	4	8	16	25	35	46	…	1120

当我们需要添加数据时，我们可以直接利用额外的空间并增加列表的有效容量，`N`。我们继续添加数据，`N` 会继续增长直到 `N == M`。此时，没有额外的空间给我们插入，我们必须创建一个拥有更多额外空间的新列表。这个新列表的额外空间大小如例 3-5 的公式所示，然后我们将旧数据复制进新的空间。

这一系列的事件见图 3-3。该图显示了例 3-6 中列表 l 的各种操作。

例 3-6　列表大小的改变

```
l = [1, 2]
for i in range(3, 7):
    l.append(i)
```

备忘

这一超额分配发生在第一次往列表里添加元素时。在一个列表被直接创建时，如前例，分配的元素数量是完全按需的。

额外分配的空间一般来说非常小，但累加起来就不可忽视。当你在维护很多小列表或一个非常大的列表时，这一效果会变得十分显著。如果我们维护 1 000 000 个列表，每个列表都包含 10 个元素，那么我们可能会假设占用了 10 000 000 个元素的内存。但是如果在构建列表时用了 append 操作，实际占用的内存可能是 16 000 000 个元素。同样，对于一个拥有 100 000 000 个元素的大列表，实际分配的可能是 112 500 007 个元素！

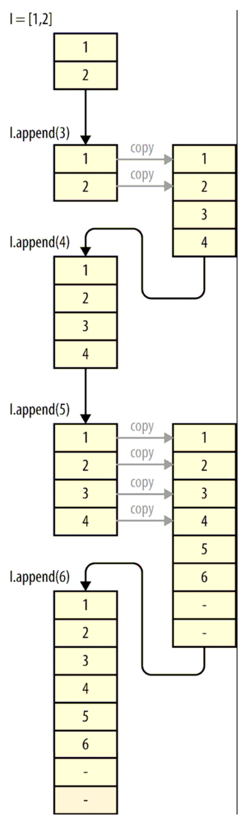


图 3-3　一个列表在多次添加时的变化示例

3.2.2 静态数组：元组

元组固定且不可变。这意味着一旦元组被创建，和列表不同，它的内容无法被修改或它的大小也无法被改变：

```
>>> t = (1,2,3,4)
>>> t[0] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

虽然它们不支持改变大小，但是我们可以将两个元组合并成一个新元组。这一操作类似列表的 `resize` 操作，但我们不需要为新生成的元组分配任何额外的空间：

```
>>> t1 = (1,2,3,4)
>>> t2 = (5,6,7,8)
>>> t1 + t2
(1, 2, 3, 4, 5, 6, 7, 8)
```

如果我们将其跟列表的 `append` 操作比较，我们会看到它的复杂度是 `O(n)` 而不是列表的 `O(1)`。这是因为对元组每添加一个新元素都会有分配和复制操作，而不是像列表那样仅在额外的空间耗尽时发生。所以，元组并没有提供一个类似 `append` 的自增操作，任意两个元组相加始终返回一个新分配的元组。

不为改变大小保存额外空间带来的好处是使用了更少的资源。一个使用过 `append` 操作的大小为 100000000 的列表实际上占用了 112500007 的元素的内存，而保存同样数据的元组始终占用 100000000 个元素的内存。这使得元组对于静态数据是一个轻量级且更好的选择。

另外，即使我们创建的列表并没有使用 `append`（也就是并没有 `append` 操作导致的额外空间），它占用的内存*依然*大于保存同样数据的元组。这是因为列表需要记住更多关于它们自身状态的信息来进行高效的 `resize`。虽然这一额外的信息很少（仅一个额外元素），如果我们有几百万个列表，累加起来也不可忽视。

元组的静态特性的另一个好处体现在一些会在 Python 后台发生的事：资源缓存。Python 是一门垃圾收集语言，这意味着当一个变量不再被使用时，Python 会将该变量使用的内存释放回操作系统，以供其他程序（或变量）使用。然而，对于长度为 1～20 的元组，即使它们不再被使用，它们的空间也不会立刻被还给系统，而是留待未来使用。这意味着当未来需要一个同样大小的新元组时，我们不再需要向操作系统申请一块内存来存放数据，因为我们已经有了预留的内存。

这看上去可能只是一个细微的好处，但实际上是元组一个很神奇的地方：它们可以被轻松迅速地创建，因为它们可以避免跟操作系统打交道，而后者很花时间。例3-7 显示了初始化一个列表比初始化一个元组慢 5.1 倍——如果是在一个循环内部，这点差别会很快累加起来！

例 3-7　初始化列表和元组的时间对比

```
>>> %timeit l = [0,1,2,3,4,5,6,7,8,9]
1000000 loops, best of 3: 285 ns per loop
>>> %timeit t = (0,1,2,3,4,5,6,7,8,9)
10000000 loops, best of 3: 55.7 ns per loop
```

3.3　小结

当你的数据有一个内在次序时，使用列表和元组作为你的数据结构可以让速度更快、开销更低。数据的内在次序使你可以回避在这些数据结构内部查询的问题：如果次序已知，那么查询操作是 `O(1)`，避免了昂贵的 `O(n)` 的线性搜索。列表可以改变大小，你需要确切知道超额分配的大小来确保数据集仍然可以被保存在内存里。另一方面，元组可以迅速被创建，且无须列表的额外开销，代价则是不可修改。在第 6.1 节中，我们会讨论如何通过为列表预分配内存来减轻频繁 `append` 给 Python 列表带来的负担，并学习一些其他的优化手段来帮助管理这些问题。

在下一章，我们将浏览字典的计算属性，它以额外的开销为代价解决了无序数据的搜索/查询问题。

第 4 章

字典和集合

读完本章之后你将能够回答下列问题

- 字典和集合各自适用于什么情况？
- 字典和集合的共同点是什么？
- 字典的开销在哪里？
- 我如何优化字典的性能？
- Python 如何使用字典记录命名空间？

如果你有一些无序数据但它们可以被唯一的索引对象来引用（任何可以被散列的类型都可以成为索引对象，索引对象通常会是一个字符串），那么集合和字典就是理想的数据结构。索引对象被称为“键”，而数据被称为“值”。字典和集合几乎一模一样，只是集合实际上并不包含值：一个集合只不过是一堆键的组合。顾名思义，集合非常适用于集合操作。

备忘

可以被散列的类型是一种同时实现了__hash__魔法函数以及__eq__或__cmp__两者之一的类型。所有的 Python 原生类型都实现了它们，而用户自定义类则都有默认的值。4.1.4 节中会介绍更多细节。

在上一章，我们看到对次序未知的列表/元组的最优查询时间是 O(log n)（使用搜索操作），而字典和集合基于键的查询则可以带给我们 O(1)（译注：原文这里为O(n)，根据上下文判断应为O(1)）的查询时间。除此之外，和列表/元组一样，

字典和集合的插入时间是 `O(1)`①。在 4.1 节中我们会看到，为了达到这一速度，它们在底层所使用的数据结构是一个开放地址散列表。

然而，使用字典和集合有其代价。首先它们通常会占用更多的内存。同时，虽然插入/查询的复杂度是 `O(1)`，但实际的速度极大取决于其使用的散列函数。如果散列函数的运行速度较慢，那么在字典和集合上进行的任何操作也会相应变慢。

让我们看一个例子。假设我们需要存储一个电话簿中每个人的联系信息，并且能够在将来轻松回答问题“John Doe 的电话号码是什么？”如果使用列表，我们会将电话号码和名字依次存储并在需要查询时检索整个列表，如例 4-1 所示。

例 4-1　列表查询电话簿

```
def find_phonenumber(phonebook, name):
    for n, p in phonebook:
        if n == name:
            return p
    return None

phonebook = [
    ("John Doe", "555-555-5555"),
    ("Albert Einstein", "212-555-5555"),
]
print "John Doe's phone number is", find_phonenumber(phonebook, "John Doe")
```

备忘

我们也可以将列表排序并用 `bisect` 模块获得 `O(log n)` 的性能。

但如果使用字典，我们只需要以名字为“键”，以电话号码为“值”，如例 4-2 所示。这让我们得以通过简单查询就获得我们需要的值的直接引用，而不是去数据集中读取每一个值。

例 4-2　字典查询电话簿

```
phonebook = {
    "John Doe": "555-555-5555",
    "Albert Einstein" : "212-555-5555",
}
print "John Doe's phone number is", phonebook["John Doe"]
```

① 我们将在 4.1.4 节中讨论，字典和集合十分依赖它们的散列函数。如果它们的散列函数对某个数据类型不具有 `O(1)` 的计算时间，那么包含该数据类型的字典和集合都不具有 `O(1)` 保证。

对大型电话簿来说，字典的 O(1)查询和列表的 O(n)线性搜索（或使用 bisect 模块后的 O(log n)）之间的区别是十分可观的。

问题

创建一个脚本来比较 bisect 列表和字典在解决电话簿查询问题上的时间。当电话簿大小增长时，时间会如何变化？

另一方面，如果我们想要回答的问题是“我的电话簿中有多少个不同的名字？”我们就可以使用集合的能力。记住集合就是一堆键的组合——这正是我们需要给数据添加的属性。这跟基于列表的解决方案是一个鲜明的对比，为了给列表加上这一属性，我们将不得不拿出每一个名字并和其他所有名字进行比较，见例 4-3。

例 4-3　用列表和集合查询不同的名字

```
def list_unique_names(phonebook):
    unique_names = []
    for name, phonenumber in phonebook:                    #❶
        first_name, last_name = name.split(" ", 1)
        for unique in unique_names:                        #❷
            if unique == first_name:
                break
        else:
            unique_names.append(first_name)
    return len(unique_names)

def set_unique_names(phonebook):
    unique_names = set()
    for name, phonenumber in phonebook:                    #❸
        first_name, last_name = name.split(" ", 1)
        unique_names.add(first_name)                       #❹
    return len(unique_names)

phonebook = [
    ("John Doe", "555-555-5555"),
    ("Albert Einstein", "212-555-5555"),
    ("John Murphey", "202-555-5555"),
    ("Albert Rutherford", "647-555-5555"),
    ("Elaine Bodian", "301-555-5555"),
]

print "Number of unique names from set method:", set_unique_names(phonebook)
print "Number of unique names from list method:", list_unique_names(phonebook)
```

❶❸ 我们必须遍历电话簿的每一项，所以这一循环的代价是 O(n)。

❷ 这里，我们必须比较当前的名字和所有已知的名字。如果它是一个新名字，则加入已知名字列表。我们继续遍历列表进行这一步骤，直到电话簿中所有的项都被遍历。

❹ 对于集合，我们只需要简单地将当前名字添加进集合，而无须遍历所有的已知名字。因为集合保证了它包含的键的唯一性，如果你尝试添加一个已有的项，该项不会被添加进集合。另外，这一操作的代价是 O(1)。

列表算法的内循环会遍历 unique_names，它从空列表开始增长，在最坏情况下，所有的名字都是唯一的，那么它最终会增长到跟电话簿一样大小。这可以被看作是在一个不断增长的列表上线性搜索电话簿中的每一个名字。因此，完整算法具有 O(n log n)的复杂度，因为外循环的贡献是 O(n)而内循环的贡献则是 O(log n)。

另一方面，集合算法没有内循环，无论电话簿多大，set.add 操作都可以在一个固定的操作次数中完成，是一个代价为 O(1)的过程（关于这点，我们在讨论字典和集合的实现时会有一些细节方面的警告）。因此，对于这一算法复杂度的唯一非常数贡献是对电话簿的遍历，整个算法的复杂度就是 O(n)。

我们用一个具有 10 000 个条目以及 7 422 个唯一名字的电话簿对这两个算法进行计时，我们会看到 O(n)和 O(n log n)能有多大的差距：

```
>>> %timeit list_unique_names(large_phonebook)
1 loops, best of 3: 2.56 s per loop

>>> %timeit set_unique_names(large_phonebook)
100 loops, best of 3: 9.57 ms per loop
```

也就是说，集合算法的速度是列表的 267 倍！另外，当电话簿大小增长时，速度的提升也在增加（对于一个具有 100 000 个条目以及 15 574 个唯一名字的电话簿，这一差距是 557 倍）。

4.1 字典和集合如何工作

字典和集合使用散列表来获得 O(1)的查询和插入。能得到这一效率是因为我们非常聪明地使用散列函数将一个任意的键（如一个字符串或一个对象）转变成了一个列表的索引。散列函数和列表随后可以被用来决定任意数据的位置，而无须搜索。通过将一个数据的键转化成某种可以被用作列表索引的东西，我们就得到了跟列表一样的性能。而且，由于无须使用数字作为索引（它本身暗示了数据的某种顺序），我们可以用任意的键来引用数据。

4.1.1 插入和获取

为了创建一个散列表，我们先从分配一些内存开始，这一点和数组一样。对于一个数组，如果我们想要插入数据，我们只需要找到未被使用的最小的桶并将我们的数据插入那里（如果有必要的话还要改变数组的大小）即可。对于散列表，我们必须首先弄清楚数据在这个连续内存块中的位置。

新插入数据的位置取决于数据的两个属性：键的散列值以及该值如何跟其他对象比较。这是因为当我们插入数据时，首先需要计算键的散列值并掩码来得到一个有效的数组索引。[①]掩码是为了保证一个可能是任意数字的散列值最终能落入分配的桶中。所以，如果我们分配了 8 个块的内存，而我们的散列值是 `28957`，那么它将落入的桶的索引是 `28957 & 0b111 = 7`。如果我们的字典增长到了需要 512 块内存，那么掩码就变成 `0b11111111`（此时，我们会使用位于 `28957 & 0b11111111` 的桶）。现在我们需要检查这个桶是否已经被使用。如果它是空桶，我们可以将键和值插入这一内存块。我们保存键是为了在获取时确保获得的是正确的值。如果桶已经被使用，且桶内的值跟我们希望插入的值相等（使用内建的 `cmp` 进行比较），说明这一键值对已经保存于散列表中，我们可以直接返回。然而，如果值不相等，那么我们需要找到一个新的位置来保存数据。

为了找到新的索引，我们用一个简单的线性函数计算出一个新的索引，这一方法称为*嗅探*。Python 的嗅探机制使用了原始散列值的高位比特（还记得对于之前那个长度为 8 的散列表，由于使用的掩码 `mask = 0b111 = bin(8 - 1)`，我们只用了最后 3 个 `bit` 作为初始索引）。使用这些高位比特使得每一个散列值生成的下一可用散列序列都是不同的，这样就能帮助防止未来的碰撞。生成新索引的算法有很多自由的选择，不过，需要注意的是计算方案要能访问每一个可能的索引使得数据能够尽量均匀地分布在表中。数据分布的均匀程度被称为“负载因素”，它跟散列函数的熵有关。例 4-4 中的伪码描述了 CPython 2.7 中使用的散列索引的计算。

例 4-4　字典查询序列

```
def index_sequence(key, mask=0b111, PERTURB_SHIFT=5):
    perturb = hash(key)  #❶
    i = perturb & mask
    yield i
    while True:
        i = ((i << 2) + i + perturb + 1)
```

① 掩码是一个二进制数，用来截断另一个数字。比如，`0b1111101 & 0b111 = 0b101 = 5` 意味着掩码 `0b111` 截断了数字 `0b1111101`。这一操作也可以被看作是获取一个数字的最低几位。

```
        perturb >>= PERTURB_SHIFT
        yield i & mask
```

❶ hash 返回的是一个整型，而 CPython 中实际的 C 代码使用的是无符号整型。因此，这段伪码并没有 100%复制 CPython 的行为，但它是一个不错的近似。

这一嗅探函数是基础的“线性嗅探”方法的修改版。在线性嗅探中，我们让 `i = (5 * i + 1) & mask`，`i` 的初始值是键的散列值，算法中用到的 5 这个值对于当前的讨论不重要。[①]重要的是记住线性嗅探仅使用了散列值的最后几个字节而没有考虑其余字节（比如，对于一个 8 元素字典，我们只查看了最后 3 个比特，因为当前的掩码是 `0x111`）。这意味着如果两个键的散列值最后 3 个比特相等，那么不仅产生了一个碰撞，同时其后的嗅探索引序列也都是一样的。Python 使用的扰动方案会考虑散列值中更多的比特位来解决这一问题。

当我们在查询某个键时也有一个类似的过程：给出的键会被转化为一个索引进行检索。如果和该索引指向的位置中的键符合（在插入操作时我们会保存原始的键），那么我们就会返回那个值。如果不符合，我们用同一个方案继续创建新的索引，直至我们找到数据或找到一个空桶。如果我们找到一个空桶，我们就可以认为表里不存在该数据。

图 4-1 显示了往散列表中添加数据的过程。我们创建的散列函数只使用输入的第一个字符。我们利用了 Python 的 `ord` 函数将输入的第一个字符转成一个整型（散列函数必须返回整型）。我们在 4.1.4 节中将会看到，Python 为其大多数类型都提供了一个散列函数，所以你无须自己提供一个，除非是在某些极端情况下。

插入键“Barcelona”时发生了一个碰撞，用例 4-4 的方案计算出一个新的索引。创建这个字典的 Python 代码见例 4-5。

例 4-5 自定义散列函数

```
class City(str):
    def __hash__(self):
        return ord(self[0])

# We create a dictionary where we assign arbitrary values to cities
data = {
    City("Rome"): 4,
    City("San Francisco"): 3,
    City("New York"): 5,
    City("Barcelona"): 2,
}
```

① 5 这个值来自线性同余生成器（LCG）的一个属性，它被用来生成随机数。

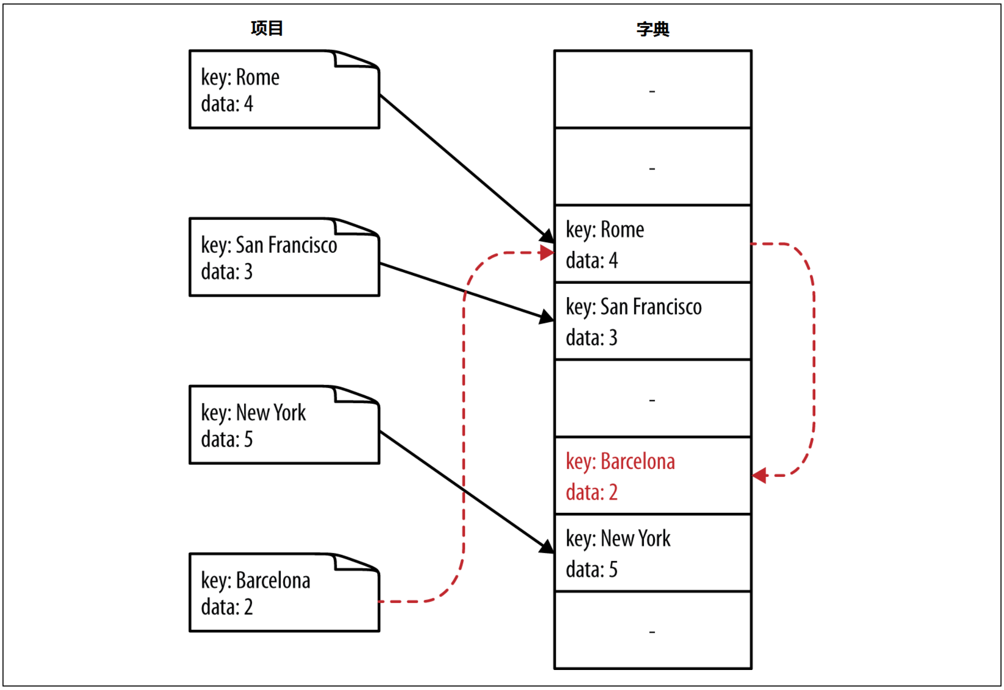


图 4-1　插入散列表时发生碰撞的结果

在这里，“Barcelona”和“Rome”发生了散列碰撞（图 4-1 显示了插入时的结果）。我们看到，对于一个拥有四个元素的字典，它的掩码是 `0b111`。“Barcelona”会尝试使用索引 `ord("B") & 0b111 = 66 & 0b111 = 0b1000010 & 0b111 = 0b010 = 2`。同样，“Rome”会尝试使用索引 `ord("R") & 0b111 = 82 & 0b111 = 0b1010010 & 0b111 = 0b010 = 2`。



问题

思考下面的问题，讨论散列碰撞：

1. 查找一个元素——使用例 4-5 创建的字典，对“Johannesburg”键的查询是怎样的？检索的索引是什么？

2. 删除一个元素——使用例 4-5 创建的字典，如何处理“Rome”键的删除？后续对“Rome”和“Barcelona”键的查询会被如何处理？

3. 散列碰撞——考虑例 4-5 创建的字典，如果有 500 个首字母大写的城市被插入散列表，你估计会有多少次散列碰撞？1000 个城市又如何？你能否想到一种方法降低碰撞次数？

> 对于 500 个城市，大约有 474 个字典元素会跟之前的值冲突（500−26），每个散列值会有大约 500 / 26 = 19.2 个城市与之关联。对于 1000 个城市则是 974 次碰撞，每个散列值有 1000 / 26 = 38.4 个城市与之关联。这是因为散列函数只是简单地基于首字母的数值，其取值范围为 `A-Z`，仅有 26 个独立散列值。这意味着一次查询会导致最多 38 次子查询来找到正确的值。为了修正这一点，我们必须考虑城市的其他方面来增加可能的散列值。默认的字符串散列函数会考虑每一个字符使散列值的可能取值范围最大化。4.1.4 节中有更多解释。

4.1.2 删除

当一个值从散列表中被删除时，我们不能简单地写一个 `NULL` 到内存的那个桶里。这是因为我们已经用 `NULL` 来作为嗅探散列碰撞的终止值。所以，我们必须写一个特殊的值来表示该桶虽空，但其后可能还有别的因散列碰撞而插入的值。这些空桶可以在未来被写入或在散列表改变大小时被删除。

4.1.3 改变大小

当越来越多的项目被插入散列表时，表本身必须改变大小来适应。研究显示一个不超过三分之二满的表在具有最佳空间节约的同时依然具有不错的散列碰撞避免率。因此，当一个表到达关键点时，它就会增长。为了做到这一点，需要分配一个更大的表（也就是在内存中预留更多的桶），将掩码调整为适合新的表，旧表中的所有元素被重新插入新表。这需要重新计算索引，因为改变后的掩码会改变索引计算结果。结果就是，改大散列表的代价非常昂贵！不过，因为我们只在表太小时而不是在每一次插入时进行这一操作，分摊后每一次插入的代价依然是 `O(1)`。

字典或集合默认的最小长度是 8（也就是说，即使你只保存 3 个值，Python 仍然会分配 8 个元素）。每次改变大小时，桶的个数增加到原来的 4 倍，直至达到 50000 个元素，之后每次增加到原来的 2 倍。这导致了下面可能的大小：

```
8, 32, 128, 512, 2048, 8192, 32768, 131072, 262144, ...
```

值得注意的是当一个散列表变大或变小时都可能发生改变大小。也就是说，如果散列表中足够多的元素被删除，表可能会被改小。但是，改变大小*仅发生在插入时*。

4.1.4 散列函数和熵

Python 对象通常以散列表实现，因为它们已经有内建的`__hash__`和`__cmp__`函数。对于数字类型（`int` 和 `float`），散列值就是基于它们数字的比特值。元组和

字符串的散列值基于它们的内容。而另一方面，列表不能被散列，因为它们的值可以改变。一旦列表的值发生改变，其散列值也会相应改变，也就改变了键在散列表中的相对位置。[1]

用户自定义类也有一个默认的散列和比较函数。默认的`__hash__`函数使用内建的`id`函数返回对象在内存中的位置。同样，`__cmp__`操作符比较的也是对象在内存中的位置的数字值。

这么做一般来说没什么问题，因为一个类的两个实例一般是不同的，不会导致散列碰撞。然而在某些情况下我们会想要使用`set`或`dict`对象来消除项目之间的歧义。见下例的类定义：

```
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y
```

如果我们用相同的`x`和`y`实例化多个`Point`对象，它们在内存中是独立的对象，各自处在内存中的不同位置，也就有各自不同的散列值。这意味着将它们放入一个`set`会导致它们各自都是一个独立的项目：

```
>>> p1 = Point(1,1)
>>> p2 = Point(1,1)
>>> set([p1, p2])
set([<__main__.Point at 0x1099bfc90>, <__main__.Point at 0x1099bfbd0>])
>>> Point(1,1) in set([p1, p2])
False
```

我们可以提供一个基于对象内容而不是对象在内存中的位置的自定义散列函数来解决这个问题。散列函数可以是任意函数，只要它对于同一个对象始终给出同一个散列值。（散列函数对熵也有要求，我们后面会加以讨论。）下面的例子重定义了Point类，给出了我们期望的结果：

```
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __hash__(self):
        return hash((self.x, self.y))

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
```

这让我们在集合或字典中创建的项目能够以`Point`对象的内部属性而不是其在内存中的位置为索引：

① 更多信息见 http://wiki.python.org/moin/DictionaryKeys。

```
>>> p1 = Point(1,1)
>>> p2 = Point(1,1)
>>> set([p1, p2])
set([<__main__.Point at 0x109b95910>])
>>> Point(1,1) in set([p1, p2])
True
```

之前我们讨论散列碰撞时提到，一个用户自定义的散列函数需要让散列值均匀分布以避免散列碰撞。碰撞太频繁会降低散列表的性能：如果大多数键都有碰撞，那么我们需要经常“嗅探”其他索引值，有可能需要在字典中访问很大一片区域来寻找需要的键。最坏情况下，字典中所有的键都碰撞，字典查询的性能变成`O(n)`，就跟搜索一个列表一样。

如果我们在创建散列函数时知道我们要在字典中存储 5000 个值，我们必须注意整个字典将被存储在一个大小为 32768 的散列表中，所以我们的散列值只有最后 15 个比特参与了索引的创建（对于这个大小的散列表，掩码是 `bin(32768-1) = 0b111111111111111`）。

衡量“我的散列函数分布均匀程度”的标准被称为散列函数的熵。熵的定义是：

$$S = -\sum_{i} p(i) \cdot \log(p(i))$$

`p(i)`是散列函数给出散列值为 `i` 的概率。当所有的散列值都具有同样的被选中概率时熵最大。一个令熵最大的散列函数被称为*完美*散列函数，因为它保证了最低的碰撞次数。

对于无穷大的字典，整数的散列函数就是完美散列函数。这是因为一个整数的散列值就是整数自身！一个无穷大字典的掩码也是无穷大，所以我们会使用散列值的所有比特。于是，给定任意两个数字，我们都可以保证它们的散列值不会相等。

不过，如果我们考虑有限大的字典，那么我们将不再具有这样的保证。比如，对于一个具有四个元素的字典，我们使用的掩码是 `0b111`。那么，数字 5 的散列值是 `5 & 0b111 = 5`，而 `501` 的散列值是 `501 & 0b111 = 5`，它们的条目会碰撞。

备忘

为了找到大小为 `N` 的字典的掩码，我们首先找到能令该字典保持三分之二满的最低桶数（`N * 5 / 3`），然后找到能满足这个数字的最小字典大小（8; 32; 128; 512; 2048; 等等）并找到足以保存这一数字的 bit 的位数。比如，如果 `N=1039`，那么我们至少需要 1731 个桶，这意味着我们的字典有 2048 个桶。那么掩码就是 `bin(2048 - 1) = 0b11111111111`。

对于有限大小的字典不存在一个最佳的散列函数。不过，提前知道散列值的范围以及字典的大小可以帮助我们做出好的选择。比如，如果我们要在字典中存储总共 676 个双小写字母组合（aa、ab、ac 等），那么例 4-6 就是一个好的散列函数。

例 4-6　最优双字母散列函数

```
def twoletter_hash(key):
    offset = ord('a')
    k1, k2 = key
    return (ord(k2) - offset) + 26 * (ord(k1) - offset)
```

当掩码为 `0b11111111111` 时（一个具有 676 个元素的字典将被保存在一个长度为 2048 的散列表中，其掩码是 `bin(2048-1) = 0b11111111111`），这个散列函数对于任意两个双小写字母都不会发生散列碰撞。

例 4-7 十分明显地展示了一个用户自定义类使用了坏的散列函数的后果——在这里，一个坏的散列函数（实际上是最糟糕的散列函数！）的代价是查询变慢 21.8 倍。

例 4-7　好坏散列函数的时间区别

```
import string
import timeit

class BadHash(str):
    def __hash__(self):
        return 42

class GoodHash(str):
    def __hash__(self):
        """
        This is a slightly optimized version of twoletter_hash
        """
        return ord(self[1]) + 26 * ord(self[0]) - 2619

baddict = set()
gooddict = set()
for i in string.ascii_lowercase:
    for j in string.ascii_lowercase:
        key = i + j
        baddict.add(BadHash(key))
        gooddict.add(GoodHash(key))

badtime = timeit.repeat(
    "key in baddict",
    setup = "from __main__ import baddict, BadHash; key = BadHash('zz')",
    repeat = 3,
    number = 1000000,
```

```
)
goodtime = timeit.repeat(
    "key in gooddict",
    setup = "from __main__ import gooddict, GoodHash; key = GoodHash('zz')",
    repeat = 3,
    number = 1000000,
)

print "Min lookup time for baddict: ", min(badtime)
print "Min lookup time for gooddict: ", min(goodtime)

# Results:
# Min lookup time for baddict: 16.3375990391
# Min lookup time for gooddict: 0.748275995255
```

问题

1. 证明对于一个有限字典（具有有限掩码），使用整数的值作为其散列值不会导致碰撞。

2. 证明例 4-6 的散列函数对于一个大小为 1024 的散列表是完美的。为什么它对更小的散列表不完美？

4.2 字典和命名空间

字典的查询很快，不过，在不必要的时候这么做会让你的代码变慢，就跟任何非必要代码行一样。Python 在命名空间的管理上就浮现出这一问题，它过度使用了字典来进行查询。

每当 Python 访问一个变量、函数或模块时，都有一个体系来决定它去哪里查找这些对象。首先，Python 查找 `locals()`数组，其内保存了所有本地变量的条目。Python 花了很多精力优化本地变量查询的速度，而这也是整条链上唯一一个不需要字典查询的部分。如果它不在本地变量里，那么会搜索`globals()`字典。最后，如果对象也不在那里，则搜索`__builtin__`对象。要注意`locals()`和`globals()`是显式的字典而`__builtin__`则是模块对象，在搜索`__builtin__`中的一个属性时，我们其实是在搜索它的 `locals()`字典（对所有的模块对象和类对象都是如此！）。

为了让事情更清楚，让我们看一个简单的例子，对不同作用域内的函数进行调用（例 4-8）。我们可以用 `dis` 模块（例 4-9）解析函数来更好地理解这些命名空间查询是怎么发生的。

例 4-8　命名空间查询

```
import math
from math import sin

def test1(x):
    """
    >>> %timeit test1(123456)
    1000000 loops, best of 3: 381 ns per loop
    """
    return math.sin(x)

def test2(x):
    """
    >>> %timeit test2(123456)
    1000000 loops, best of 3: 311 ns per loop
    """
    return sin(x)

def test3(x, sin=math.sin):
    """
    >>> %timeit test3(123456)
    1000000 loops, best of 3: 306 ns per loop
    """
    return sin(x)
```

例 4-9　命名空间查询解析

```
>>> dis.dis(test1)
  9           0 LOAD_GLOBAL              0 (math) # Dictionary lookup
              3 LOAD_ATTR                1 (sin)  # Dictionary lookup
              6 LOAD_FAST                0 (x)    # Local lookup
              9 CALL_FUNCTION            1
             12 RETURN_VALUE

>>> dis.dis(test2)
  15          0 LOAD_GLOBAL              0 (sin) # Dictionary lookup
              3 LOAD_FAST                0 (x)   # Local lookup
              6 CALL_FUNCTION            1
              9 RETURN_VALUE

>>> dis.dis(test3)
  21          0 LOAD_FAST                1 (sin) # Local lookup
              3 LOAD_FAST                0 (x)   # Local lookup
              6 CALL_FUNCTION            1
              9 RETURN_VALUE
```

第一个函数 test1 显示查询 math 库来调用 sin。生成的字节码的证据表明：首先一个 math 模块的引用必须被调入，然后在模块上进行属性查询，直到找到 sin 函数。整个步骤经过了两次字典查询，一次查找 math 模块，一次在模块中查找 sin 函数。

另一方面，test2 从 math 模块显式导入了 sin 函数，因此该函数可在全局命名空间中被直接访问。这意味着我们可以避免查询 math 模块以及后续的属性查询。不过，我们还是要在全局命名空间查找 sin 函数。这也是另一个我们要从模块中显式导入函数的原因。这样做不仅让代码更可读，因为读者可以知道到底需要外部资源中的什么函数，而且也加速了代码！

最后，test3 定义了 sin 函数为一个参数关键字，其默认值是 math 模块的 sin 函数的引用。虽然我们依然需要在模块中查找这一函数，但仅在 test3 函数第一次被定义时查找。之后，这一引用以默认参数关键字的形式作为一个本地变量被保存在函数的定义中。之前提到过，本地变量无须字典查询；它们被保存在一个十分微小的数组中，具有很快的查询速度。因此，找到这个函数非常快。

这些效果只是 Python 对命名空间管理方式的一个有趣结果，test3 并不是 Python 的惯用写法。幸运的是，这些额外的字典查询仅在它们被大量调用时才会开始降低性能（比如朱利亚集合例子中在一个高速循环的最内部）。记住这一点，一个更可读的解决方案是在循环开始前设置一个本地变量保存一个函数的全局引用。在调用函数时我们依然会进行一次全局查询，但在循环内对函数的每次调用都会变快。考虑到代码中即使是十分微小的减慢也会被数百万次的运行所放大，即使一次字典查询仅需花费几百纳秒，如果我们循环几百万次这样的查询，那总耗时就会迅速累加。事实上，例 4-10 的查询中，我们可以看到只需在循环前将 sin 函数本地化就能获得 9.4%的速度提升。

例 4-10　循环内的命名空间查询的降速效果

```
from math import sin

def tight_loop_slow(iterations):
    """
    >>> %timeit tight_loop_slow(10000000)
    1 loops, best of 3: 2.21 s per loop
    """
    result = 0
    for i in xrange(iterations):
        # this call to sin requires a global lookup
        result += sin(i)

def tight_loop_fast(iterations):
    """
    >>> %timeit tight_loop_fast(10000000)
    1 loops, best of 3: 2.02 s per loop
    """
    result = 0
    local_sin = sin
```

```
    for i in xrange(iterations):
        # this call to local_sin requires a local lookup
        result += local_sin(i)
```

4.3 小结

字典和集合适用于存储能够被键索引的数据。散列函数对键的使用方式极大地影响数据结构的最终性能。另外，理解字典如何工作不仅可以让你更好地组织你的数据，同时也能让你更好地组织你的代码，因为字典是 Python 的内部功能之一。

我们将在下一章探索生成器，它令我们能够对数据进行更强的控制，而且不需要预先在内存中保存完整的数据集。这让我们得以绕过很多使用 Python 内部数据结构时可能遇到的障碍。

第 5 章

迭代器和生成器

读完本章之后你将能够回答下列问题

- 生成器是怎样节约内存的?
- 使用生成器的最佳时机是什么?
- 我如何使用 `itertools` 来创建复杂的生成器工作流?
- 延迟估值何时有益,何时无益?

当许多其他语言的使用者开始学习 Python 时,他们在 `for` 循环的记法差异上吓了一跳。这是因为,在其他语言中的写法可能是这样:

```
# 其他语言
for (i=0; i<N; i++) {
    do_work(i);
}
```

而在 Python 中他们遇到了一个叫 `range` 或 `xrange` 的新函数:

```
# Python
for i in range(N):
    do_work(i)
```

这两个函数让我们见识到了使用生成器编程的范例。为了彻底理解生成器,让我们首先试着简单实现 `range` 和 `xrange` 函数:

```
def range(start, stop, step=1):
    numbers = []
    while start < stop:
        numbers.append(start)
        start += step
```

```
    return numbers

def xrange(start, stop, step=1):
    while start < stop:
        yield start #❶
        start += step

for i in range(1,10000):
    pass

for i in xrange(1,10000):
    pass
```

❶ 这个函数会 yield 很多值而不是只返回一个。这就让一个看上去很普通的函数转变成一个生成器，一种可以被重复轮询下一个可用值的函数。

首先要注意的是 range 的实现必须预先创建一个列表保存范围内的所有数字。所以，如果范围从 1 到 10000，这个函数会对 numbers 列表进行 10000 次 append（在第 3 章我们已经讨论过这部分开销），然后返回整个列表。而另一方面，生成器则能够"返回"很多值。每次代码运行到 yield，该函数都会发射出一个值，然后当外部代码需求另一个值时，该函数才会继续运行（之前的状态保持不变）并发射新的值。当该函数运行结束时，一个 StopIteration 异常会被抛出表明该生成器已经没有更多的值了。结果就是，虽然两个函数最终运行了同样的计算次数，使用 range 版本的循环多消耗了 10000 倍的内存（如果范围从 1 到 N 则是 N 倍）。

牢记这段代码，我们就能在 for 循环中使用自己实现的 range 和 xrange。在 Python 语言中，for 循环要求被循环的对象支持迭代。这意味着我们必须能够在循环对象上创建一个迭代器。而在任何对象上创建迭代器，我们都只需要使用 Python 内建的 iter 函数。这个函数，对于列表、元组、字典和集合都返回一个对象内部元素或键的迭代器。对于更复杂的对象，iter 函数会返回对象的__iter__属性。由于 xrange 已经返回了一个迭代器，对其调用 iter 就不会做任何事，只是简单返回原始的对象（因此 type(xrange(1, 10)) == type(iter(xrange(1, 10))))。不过，由于 range 返回的是一个列表，我们就不得不创建一个新的对象，一个列表的迭代器，来迭代列表中的所有值。一旦一个迭代器被创建，我们只需要对其调用 next()函数，来获得新值，直到 StopIteration 异常被抛出。这给我们提供了一个能够很好解析 for 循环的视图，如例 5-1 所示。

例 5-1　Python 的 for 循环解析

```
# python 循环
for i in object:
```

```
    do_work(i)

# 等同于
object_iterator = iter(object)
while True:
    try:
        i = object_iterator.next()
        do_work(i)
    except StopIteration:
        break
```

for 循环的代码显示了我们在使用 range 而不是 xrange 时需要额外调用 iter。在使用 xrange 时，我们不需要做任何事情（因为它已经是一个迭代器了！）；然而在使用 range 时，我们需要分配一个新的列表并预先计算其内的值，然后我们还要创建一个迭代器！

更重要的是，预先计算 range 列表需要为整个数据集分配足够的空间并为每一个元素赋正确的值，即使我们每次仅需要一个值。为列表分配的空间其实并没有什么意义。实际上，循环甚至根本无法运行，因为 range 尝试去分配的内存可能根本无法被满足（range(100,000,000) 会创建一个 3.1GB 大小的列表！）。通过对结果计时，我们可以很明显地看到这一点：

```
def test_range():
    """
    >>> %timeit test_range()
    1 loops, best of 3: 446 ms per loop
    """
    for i in range(1, 10000000):
        pass

def test_xrange():
    """
    >>> %timeit test_xrange()
    1 loops, best of 3: 276 ms per loop
    """
    for i in xrange(1, 10000000):
        pass
```

这个问题现在看上去可能并不算什么——我们只需要将所有的 range 调用替换成 xrange——但是实际的问题可能隐藏在非常深的角落。比如假设我们有一个数字的长列表，我们想要知道其中有多少个数字可以被 3 整除。你可能会这样写：

```
divisible_by_three = len([n for n in list_of_numbers if n % 3 == 0])
```

然而，这种写法的问题跟 range 一样。因为我们使用了列表表达式，我们预先生成了一个能被 3 整除的数字的列表，仅仅只是为了对其进行计算并丢弃。如果这个

列表很长，这可能会导致无意义地分配大量内存——大到根本无法被满足。

我们的列表表达式使用了如下公式[<value> for <item> in <sequence> if <condition>]。这会创建一个列表，内含所有满足条件的值。相对的，我们也可以使用一个类似的语法(<value> for <item> in <sequence> if <condition>)来创建一个满足条件的值的生成器而不是一个列表。

利用列表表达式和生成器表达式之间的这种细微的差别,我们就能对divisible_by_three 的代码进行优化。然而，生成器并没有一个 length 属性，所以我们必须要干的聪明点：

```
divisible_by_three = sum((1 for n in list_of_numbers if n % 3 == 0))
```

这里，我们的生成器在每当遇到一个能被 3 整除的数字时就会发射一个 1，而不是别的数字。对该生成器发射的所有值求和，我们就能得到跟列表表达式一样的结果。这两个版本的代码性能几乎一样，但是生成器表达式需要的内存远小于列表表达式。另外，我们可以将列表版本轻易转化成生成器版本的原因是我们只关心列表中每个元素的当前值——该值要么可以被 3 整除要么不可以，而跟它在列表中的位置无关，跟其前后的数字也无关。更复杂的函数也可以被转成生成器，但是根据它们的复杂度，这种转换有可能会比较困难。

5.1 无穷数列的迭代器

如果我们只需要保存有限个状态并只发射当前值，生成器可以被用来产生无穷数列。斐波那契数列就是一个很好的例子——它是一个具有两个状态变量（最后两个斐波那契数）的无穷数列：

```
def fibonacci():
    i, j = 0, 1
    while True:
        yield j
        i, j = j, i + j
```

这里可以看到，虽然 j 是那个被发射的值，我们依然需要保留 i 的记录，因为它保存了斐波那契数列的状态。生成器计算所需要的状态的数量非常关键，因为它最终会被转化成对象的内存足迹。如果我们有一个函数需要使用很多的状态，却输出很少的数据，那么使用预先计算的列表可能比生成器更合适。

生成器并没有想象中使用的那么广泛的一个原因是它们的逻辑可以被包含在你的代码中。这意味着生成器其实是用更聪明的循环组织你代码的一种方式。比如，我

们可以有多种方式回答问题“5000 以内的斐波那契数中有几个奇数？”：

```
def fibonacci_naive():
    i, j = 0, 1
    count = 0
    while j <= 5000:
        if j % 2:
            count += 1
        i, j = j, i + j
    return count

def fibonacci_transform():
    count = 0
    for f in fibonacci():
        if f > 5000:
            break
        if f % 2:
            count += 1
    return count

from itertools import islice
def fibonacci_succinct():
    is_odd = lambda x : x % 2
    first_5000 = islice(fibonacci(), 0, 5000)
    return sum(1 for x in first_5000 if is_odd(x))
```

所有这些方法都有近似的运行时属性（也就是说它们都有相同的内存足迹和同样的性能），但 fibonacci_transform 函数具有多个优势。首先，它看上去比 fibonacci_succinct 直观很多，可以轻易被另一个开发者调试和理解。关于后者的警告见下一节，我们会讨论一些 itertools 的常见用法——该模块在大大简化了很多迭代器的简单操作的同时也会迅速让 Python 代码变得不那么像 Python。另一方面，fibonacci_naive 一次做了多件事情，隐藏了它实际的计算逻辑！而使用斐波那契数列的生成器函数则不会阻碍我们理解实际的计算逻辑。最后，fibonacci_transform 可以变得更加通用。这个函数可以被重命名为 num_odd_under_5000 并接受一个生成器参数，这样它就可以工作在任意数列上。

Fibonacci_transform 函数的最后一个好处是它标注了计算的两个阶段：数据的生成和数据的转化。该函数很明显是在进行数据的转化，而 fibonacci 函数则是生成数据。这一界限的划分增加了额外的清晰度和功能：我们可以让一个转化函数工作在一组新的数据上，或在一组数据上进行多个转化。这一规范在创建复杂程序时十分重要，而使用生成器则可以明确地表示：生成器用于创建数据，而普通函数则操作生成的数据。

5.2 生成器的延迟估值

之前提到，生成器之所以能够节约我们的内存是因为它只处理当前感兴趣的值。在我们计算的任意点，我们都只能访问当前的值，而无法访问数列中的其他元素（这种算法我们通常称为“单通”或“在线”）。有时候这会令生成器难以被使用，不过有很多模块和函数可以帮助解决这一问题。

我们主要关注标准库中的 `itertools` 库。它提供了 Python 内建函数 `map`、`reduce`、`filter` 和 `zip` 的生成器版本(在 `itertools` 中分别是 `imap`、`ireduce`、`ifilter` 和 `izip`)，以及其他很多有用的函数。特别值得注意的有：

islice

　　允许对一个无穷生成器进行切片。

chain

　　将多个生成器链接到一起。

takewhile

　　给生成器添加一个终止条件。

cycle

　　通过不断重复将一个有穷生成器变成无穷。

让我们创建一个使用生成器来分析大数据集的例子。假设我们现在有一个分析函数用于处理时间点数据，数据每秒会产生一个，对于过去的 20 年——我们有 631 152 000 个数据点！该数据被保存在一个文件中，每秒生成一行，我们无法将整个数据集读入内存。如果我们想要进行一些简单的异常检测，我们可以使用生成器而无须分配任何列表！

我们需要解决的问题是：对于一个格式为“timestamp, value”的数据文件，需要找到所有含有异常值的日期，比该日均值超出 3 倍标准差之外的数字被视为异常值。我们首先写读取文件的代码，文件以一行接着一行的方式读取，然后将每一行的值输出为一个 Python 对象。我们还会创建一个 `read_fake_data` 生成器用于生成伪造数据来测试我们的算法。对于这个函数，我们依然会提供一个 `filename` 参数，来让它具有和 `read_data` 相同的函数签名，不过该参数会被丢弃。这两个函数，如例 5-2 所示，使用了延迟估值——我们仅当生成器的 `next()` 属性被调用时

才会去读取文件的下一行，或生成新的伪造数据。

例 5-2　延迟读取数据

```
from random import normalvariate, rand
from itertools import count

def read_data(filename):
    with open(filename) as fd:
        for line in fd:
            data = line.strip().split(',')
            yield map(int, data)

def read_fake_data(filename):
    for i in count():
        sigma = rand() * 10
        yield (i, normalvariate(0, sigma))
```

现在，我们可以使用 itertools 提供的 groupby 函数将同一天的 timestamp 分在一组(例 5-3)。这个函数的输入参数是一组数据和一个用于分组的关键字函数。返回的则是一个生成器，生成的每一个值都是一个元组，元组内包含该组的关键字和该组所有元素的生成器。对于关键字函数，我们会创建一个 lambda 函数返回一个 date 对象。对于同一天的数据会产生相同的 date 对象，这样就能以天来分组。这个关键字函数可以是任何东西——我们可以以小时、年或数据值的某个属性来分组。唯一的限制是数据必须是连续的。也就是说，如果我们有一个输入是 A A A A B B A A 并以字母分组，我们就会得到 3 个组，分别是(A, [A, A, A, A])、(B, [B, B])以及(A, [A, A])。

例 5-3　对我们的数据分组

```
from datetime import date
from itertools import groupby

def day_grouper(iterable):
    key = lambda (timestamp, value) : date.fromtimestamp(timestamp)
    return groupby(iterable, key)
```

现在来进行实际的异常检测。我们会遍历一天内的值并记录平均值和最大值。平均值的计算会使用一个在线的平均值和标准差算法。[①]保留最大值是因为它将是我们异常数据的最佳代表——如果最大值比平均值的 3sigma 大，那么我们就返回代表这一天的 date 对象。否则，我们返回 False。不过，我们也可以直接终止函数

① 我们使用 Knuth 的在线平均数算法。这让我们能够使用一个临时变量同时计算出平均数和一阶矩（也就是标准差）。我们还可以稍稍修改公式并添加更多的状态变量来计算高阶矩的值（每个阶一个变量）。

(并返回 None)。我们输出这些值是因为这个 check_anomaly 函数被定义为一个数据过滤器——一种对需要保留的数据返回 True 而对需要丢弃的数据返回 False 的函数。这让我们可以过滤原始的数据集并只保留符合我们条件的日子。check_anomaly 函数如例 5-4 所示。

例 5-4　基于生成器的异常检测

```
import math

def check_anomaly((day, day_data)):
    # We find the mean, standard deviation, and maximum values for the day.
    # Using a single-pass mean/standard deviation algorithm allows us to only
    # read through the day's data once.
    n = 0
    mean = 0
    M2 = 0
    max_value = None
    for timestamp, value in day_data:
        n += 1
        delta = value - mean
        mean = mean + delta/n
        M2 += delta*(value - mean)
        max_value = max(max_value, value)
    variance = M2/(n - 1)
    standard_deviation = math.sqrt(variance)

    # Here is the actual check of whether that day's data is anomalous. If it
    # is, we return the value of the day; otherwise, we return false.
    if max_value > mean + 3 * standard_deviation:
        return day
    return False
```

这个函数一个看上去可能有点奇怪的地方是参数定义中额外的一组小括号。这不是打字错误，而是因为该函数的输入来自 groupby 生成器。记住 groupby 返回的元组会成为这个 check_anomaly 函数的参数。因此，我们必须进行元组展开来正确获取组的关键字和组的数据。由于我们使用了 ifilter，另一种不需要在函数定义中进行元组展开的处理方式是定义 istarfilter,类似于 istarmap 对 imap 做的处理那样（更多信息参见 itertools 文档)。

最后，我们可以将生成器链接来获得所有具有异常数据的日子（例 5-5)。

例 5-5　将我们的生成器链接起来

```
from itertools import ifilter, imap

data = read_data(data_filename)

data_day = day_grouper(data)
```

```
anomalous_dates = ifilter(None, imap(check_anomaly, data_day)) #❶

first_anomalous_date, first_anomalous_data = anomalous_dates.next()
print "The first anomalous date is: ", first_anomalous_date
```

❶ ifilter 会移除所有不满足给定过滤器的元素。默认情况（第一个参数为 None）下，ifilter 会过滤掉所有被估值为 False 的元素。这样我们就不会包含那些 check_anomaly 认为不存在异常的日子。

这个方法非常简单地允许我们获得异常日子的列表而不需要读取整个数据集。需要注意的一件事是这段代码并没有真正进行任何计算，它只是为计算逻辑设置了一条流水线。在我们执行 anomalous_dates.next() 或以某种方式对 anomalous_dates 进行遍历之前，文件永远不会被读取。事实上，如果我们的整个数据集中有 5 个异常日子，而我们的代码在读取了第一个之后就停止，那么文件就只会被读到第一个日子的数据出现为止。这就叫延迟估值——只有被明确要求的计算才会被执行，如果出现了一个提前终止的条件，那么就可以大幅降低整体的运行时间。

用这种方式组织代码的另一个好处是它允许我们在不需要重写大部分代码的情况下轻松改换更复杂的计算逻辑。比如，如果我们想要一个日内移动窗口的分组而不是按日分组，我们可以简单写一个新的 day_grouper 来替换：

```
from datetime import datetime

def rolling_window_grouper(data, window_size=3600):
    window = tuple(islice(data, 0, window_size))
    while True:
        current_datetime = datetime.fromtimestamp(window[0][0])
        yield (current_datetime, window)
        window = window[1:] + (data.next(),)
```

现在，我们只需用这个 rolling_window_grouper 简单替换例 5-5 中的 day_grouper 就能获得我们想要的结果。使用这种模式，我们还能看到这两个方法都具有十分清晰的内存保证——它将仅保存窗口内的数据作为其状态（两个方法分别需要保存一整日的数据或 3600 个数据点）。如果就连数据集的这部分采样都依然无法全部放入内存，那么我们还可以通过多次打开文件并使用不同的文件描述符来指向我们实际需要的数据（或使用 linecache 模块）来解决这个问题。

最后一点：在 rolling_window_grouper 函数中，我们对 window 列表进行了很多 pop 和 append 操作。我们可以使用 collections 模块的 deque 对象来大大优化这部分代码。这个对象可以在列表的头部和尾部均提供 O(1) 的添加和删除

操作（传统的列表只能对列表尾部提供 O(1)的添加删除操作，而对列表的头部进行同样的操作则是 O(n)）。使用 deque 对象，我们可以使用 append 将新数据添加到列表的右侧（或者说尾部），并使用 deque.popleft()来删除列表左侧（或者说头部）的数据而无须分配更多空间或进行耗时的 O(n)操作。

5.3 小结

通过使用迭代器组织我们的异常检测算法，我们能处理的数据就远远超过了内存的限制。另外，使用迭代器的性能也好于使用列表，因为我们避免了昂贵的 append 操作。

因为迭代器是 Python 的原始类型，我们应该始终使用这个方法来减少应用程序的内存足迹。这么做带来的好处是结果可以被延迟估值，你只需要处理必需的数据，而内存就会被节省下来，因为我们不需要保存之前的结果，除非我们明确知道需要它们。在第 11 章，我们会讨论其他一些适用于更独特的问题的方法并介绍一些新的思路来解决内存问题。

我们将在第 9 章和第 10 章看到，使用迭代器来解决问题的另一个好处是它让你的代码能够适用于多 CPU 或多计算机的场合。我们在 5.1 节中讨论过，使用迭代器必须预先考虑算法所需的各种状态变量。只要你封装好了这些算法运行时必需的状态变量，算法跑在什么数据上就变得无关紧要了。比如我们在使用 multiprocessing 和 ipython 模块时就可以看到这种示范，它们都使用了一个类似于 map 的函数来启动并发任务。

第 6 章

矩阵和矢量计算

读完本章之后你将能够回答下列问题

- 矢量计算的瓶颈在哪里？
- 我可以用什么工具查看 CPU 进行计算时的效率？
- `numpy` 为什么比纯 Python 更适合数值计算？
- `cache-miss` 和 `page-faults` 是什么？
- 我如何追踪代码中的内存分配？

无论你尝试在计算机上解决什么问题，你都会在某个时候遇到矢量计算。矢量计算是计算机工作原理不可或缺的部分，也是在芯片层次上对程序运行时间进行加速所必须了解的部分——计算机只知道如何对数字进行操作，而了解如何同时进行多个这样的计算能够加速你的程序运行。

在本章，我们通过解决一个相对简单的数学问题，扩散等式求解，来揭示这个问题的复杂度并理解 CPU 层面上发生了什么。通过理解 Python 代码如何影响 CPU 以及如何有效探测这些影响，我们就可以举一反三地学习如何理解其他问题。

我们将首先介绍扩散等式问题并提供一个纯 Python 的快速解决方案。我们随后会指出该方案中的一些内存问题并试图用纯 Python 解决它们，我们会介绍 `numpy` 并验证它是如何以及为什么能够加速我们的代码的。然后我们会开始进行一些算法的改变并特化我们的代码来解决手上的问题。通过移除我们所使用的库中的一些通用性函数，我们就能够再一次提升速度。最后我们会介绍一些额外的模块，它们将帮

助我们在实际工作中简化这种流程，然后浏览一个小故事，告诫我们不要急于在性能分析之前进行优化。

6.1 问题介绍

备忘

本节要对我们将在本章解决的等式问题给予一个深刻的理解。本节的理解对于本章剩余部分的阅读并不是严格必须的。如果你想要跳过本节，那么请一定要看一下例 6-1 和例 6-2 中的算法逻辑，理解我们需要优化的代码。

另外，如果你阅读了本节之后还需要更多的解释，请阅读剑桥大学出版社出版的 William Press 等人的著作《数值分析方法库》第三版第 17 章。

为了在本章探索矩阵和矢量计算，我们将重复使用流体扩散的例子。扩散是流体移动并试图均匀混合的一种方式。

本节我们将探索扩散等式背后的数学问题。它看上去可能很复杂，但不要担心！我们会快速简化并让它更容易理解。另外，重要的是要记住，虽然对于我们所需要解决的最终等式的基本理解有助于本章的阅读，但这并不是完全必须的。后续的章节会主要集中于代码中的各种方程式，而不是扩散等式本身。不过，对等式的理解会有助于你的直觉来对代码进行优化。总而言之——理解你代码背后的动机以及算法的难点会让你对可能的优化方案有一个更深刻的理解。

扩散的一个简单例子是水中的染料：如果你在室温的水里滴入几滴染料，它会慢慢扩散直到完全跟水混在一起。由于我们并不搅拌水，水也没有热到出现热对流的程度，扩散将是两种液体混合的主要过程。为了在数学上解决这些等式，我们会特地选取我们所需要的初始条件，之后我们会看看对初始条件的改变会对计算发生怎样的影响（图 6-2）。

说了这么多，对于我们的目标来说其实最主要的还是扩散方程式本身。扩散等式可以被写成一个 1 阶偏微分方程：

$$\frac{\partial}{\partial t}u(x,t) = D\frac{\partial^2}{\partial x^2}u(x,t)$$

这个方程中，`u` 是表示扩散质量的矢量。比如，我们会有一个矢量，里面的值 `0` 表示纯水，`1` 表示纯染料（`0` 和 `1` 之间的值表示混合液体）。一般来说，这个矢量应

该是一个 2 阶或 3 阶的矩阵表示液体的实际区域或体积。这样，为了表示一个玻璃杯中的液体，我们就可以设 `u` 为一个 3 阶矩阵，计算所有的轴，而不只是在 `x` 方向求导。除此以外，`D` 是一个物理值表示模拟实验中的液体的属性。`D` 越大则表明液体越容易扩散。为了简化，我们在代码中设 `D = 1`，但依然将它包含在计算中。

备忘

扩散方程也被称为热方程。此时，`u` 表示一个区域的温度而 `D` 描述了材料的热传导能力。解开方程可以告诉我们热如何传导。这样，我们就能够了解 CPU 产生的热量如何扩散到散热片上而不是水中染料的扩散。

我们会将时空上连续的扩散微分方程改成小块的离散时空。我们用欧拉法来做到这一点。欧拉法将微分方程写成差分形式，如下：

$$\frac{\partial}{\partial t}u(x,t)=\frac{u(x,t+\mathrm{d}t)+u(x,t)}{\mathrm{d}t}$$

此时 `dt` 是一个固定的值，表示一个时间的步长，或者说我们为了解决方程而定的时间的精确度。它可以被想象成我们正在制作的电影的帧率。当帧率提高（或者说 `dt` 变小）时，我们就能够得到一个更清晰的图像。实际上，`dt` 越接近 `0`，欧拉近似就越精确（但是注意，这个精确只能在理论上达到，因为计算机只拥有有限的精确度，而精度误差会迅速传播到所有的计算结果）。这样我们就能将方程改写成根据 `u(x, t)`求 `u(x, t+dt)`。这意味着我们可以以某个初始状态 `u(x, 0)`为起点，表示一杯刚加了一滴染料的水，然后通过对初始状态进行“演化”来了解这杯水在未来某个时刻的样子（`u(x, dt)`）。这种类型的问题被称为“初值问题”或“柯西问题”。

以同样的有限差分近似对 `x` 进行求导，我们就能获得最终的方程式：

$$u(x,t+\mathrm{d}t)=u(x,t)+\mathrm{d}t*D*\frac{u(x+\mathrm{d}x,t)+u(x+\mathrm{d}x,t)+2\cdot u(x,t)}{\mathrm{d}x^2}$$

这里，和 `dt` 表示帧率一样，`dx` 表示图像的分辨率——`dx` 越小，则我们矩阵的每一个单元格表示的区域就越小。为了简化，我们设 `D = 1` 且 `dx = 1`。这两个值在物理模拟时非常重要，但是因为我们只是以扩散方程为例子来说明问题，它们对我们并不重要。

我们能用这个等式解决几乎所有扩散问题。但是，这个等式有些需要考虑的地方。首先，我们之前说过 `u` 方向的空间索引（参数 `x`）代表矩阵中的索引。那么当 `x` 处

于矩阵的起始索引位置时 x-dx 的值代表什么呢？这个问题被称为*边界条件*。你可以这样定义固定的边界条件："任何超出矩阵边界的值都为 0"（或任意其他值）。或者，你也可以定义边界值环绕的周期性边界条件（比如，如果你矩阵的某个维度的长度为 N，那么索引为-1 的值就等于索引为 N-1 的值，而索引为 N 的值则等于索引为 0 的值。换句话说，当你试图访问索引 i 上的值时，你得到的就是位于索引(i%N)上的值)。

另一个需要考虑的地方是我们如何存储 u 的多个时间分量。我们可以让一个矩阵来保存计算所需的所有时间值。看起来我们需要至少两个矩阵：一个保存液体的当前状态，一个保存液体的下一状态。我们会看到这里面有很重要的性能考量。

那么我们如何解决实际的问题？例 6-1 包含了一些伪代码来说明我们是如何使用这个等式来解决问题的。

例 6-1　1 阶扩散方程伪代码

```
# Create the initial conditions
u = vector of length N
for i in range(N):
    u = 0 if there is water, 1 if there is dye

# Evolve the initial conditions
D = 1
t = 0
dt = 0.0001
while True:
    print "Current time is: %f" % t
    unew = vector of size N

    # Update step for every cell
    for i in range(N):
        unew[i] = u[i] + D * dt * (u[(i+1)%N] + u[(i-1)%N] - 2 * u[i])
    # Move the updated solution into u
    u = unew

    visualize(u)
```

这段代码会根据水中染料的初始条件来告诉我们每一个 0.0001 秒之后的未来系统是什么样的。结果见图 6-1，图中显示的是最浓的染料液滴（高帽函数）随时间的演化。我们可以看到染料是如何均匀混合，直到每处染料的浓度都相等的。

为了本章的目的，我们还将解决方程的 2 阶版本。这意味着我们将要操作一个 2 阶矩阵，而不是一个矢量（或者说一个只有 1 阶的矩阵）。对方程唯一的改动（以及相应的代码改动）是现在我们必须计算第二个 y 方向上的导数。这意味着原始的方

程会变成：

$$\frac{\partial}{\partial t}u(x,y,t) = D \cdot \left(\frac{\partial^2}{\partial x^2}u(x,y,t) + \frac{\partial^2}{\partial y^2}u(x,y,t) \right)$$

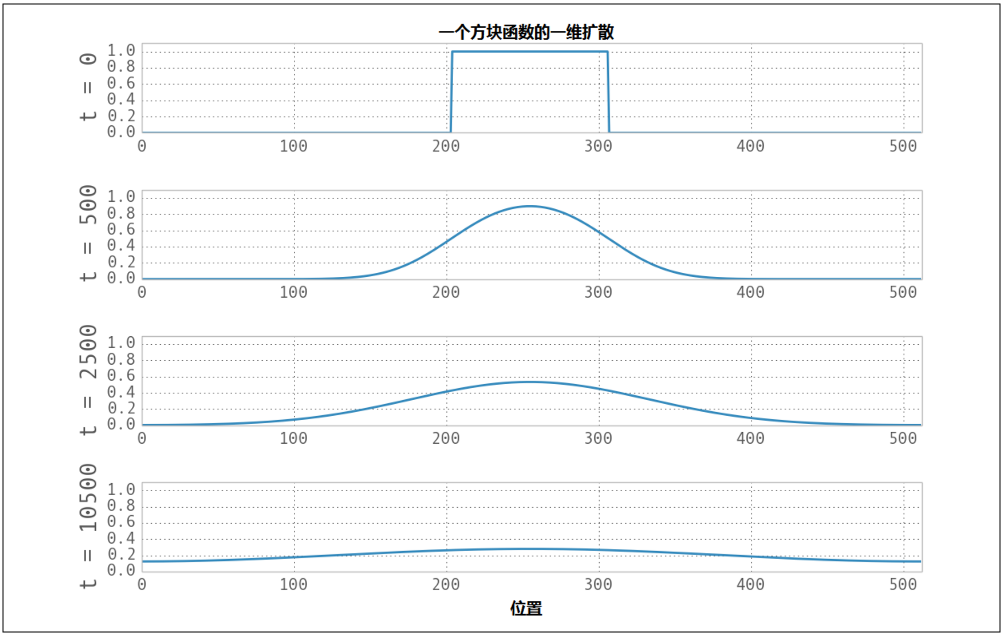


图 6-1　1 阶扩散的例子

这个 2 阶扩散方程翻译成伪码如例 6-2 所示，用的是跟之前一样的方法。

例 6-2　计算 2 阶差分的算法

```
for i in range(N):
    for j in range(M):
        unew[i][j] = u[i][j] + dt * (                          \
            (u[(i+1)%N][j] + u[(i-1)%N][j] - 2 * u[i][j]) + \ # d^2 u / dx^2
            (u[i][(j+1)%M] + u[j][(j-1)%M] - 2 * u[i][j]) \ # d^2 u / dy^2
        )
```

现在我们可以将所有的代码片段组装到一起，并写出完整的 Python2 阶扩散代码来作为我们本章接下来进行性能参照的基准。代码虽然看上去更复杂，结果却近似 1 阶扩散（见图 6-2）。

如果你希望阅读更多本节的主题，可以参考扩散方程的维基百科和 S.V.Gurevich 写的第 7 章“复杂系统的数值方法”。

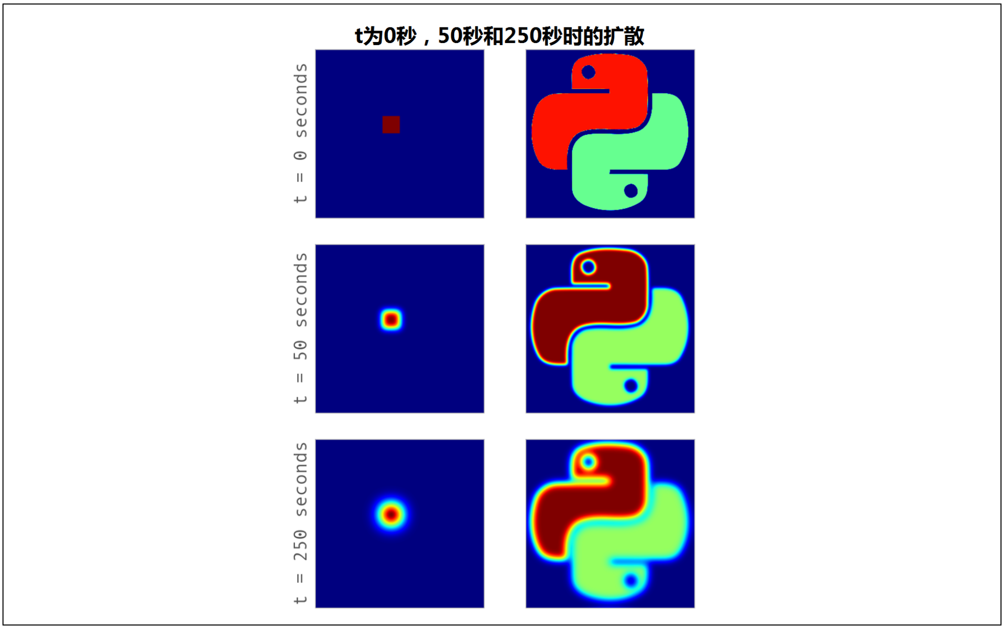


图 6-2　两组初始条件扩散的例子

6.2　Python 列表还不够吗

让我们将例 6-2 的伪代码写成正式代码，这样我们就能够更好地分析它的运行时性能。第一步是写出演化函数，它接受一个矩阵并返回其演化后的状态，见例 6-3。

例 6-3　纯 Python 2 阶扩散函数

```
grid_shape = (1024, 1024)

def evolve(grid, dt, D=1.0):
    xmax, ymax = grid_shape
    new_grid = [[0.0,] * ymax for x in xrange(xmax)]
    for i in xrange(xmax):
        for j in xrange(ymax):
            grid_xx = grid[(i+1)%xmax][j] + grid[(i-1)%xmax][j] - 2.0 * grid[i][j]
            grid_yy = grid[i][(j+1)%ymax] + grid[i][(j-1)%ymax] - 2.0 * grid[i][j]
            new_grid[i][j] = grid[i][j] + D * (grid_xx + grid_yy) * dt
return new_grid
```

备忘

其实相比预分配一个 new_grid 列表，还不如在 for 循环中用 appends 建立它。不仅速度比我们这样写要快很多，结果也是一样的。我们之所以使用现在这个方法是因为它能解释得更清楚。

全局变量 grid_shape 表示我们模拟的区域有多大。另外，在 6.1 节中解释过，我们使用的是周期性边界条件（这也是为什么我们需要对索引取模）。为了实际使用这段代码，我们必须初始化一个矩阵并对其调用 evolve。例 6-4 中的代码是一个非常通用的初始化过程，将在本章多次被重用（它的性能将不会被分析，因为它只运行一次，不像 evolve 函数将被重复调用）。

例 6-4　纯 Python 2 阶扩散初始化函数

```
def run_experiment(num_iterations):
    # Setting up initial conditions  ❶
    xmax, ymax = grid_shape
    grid = [[0.0,] * ymax for x in xrange(xmax)]

    block_low = int(grid_shape[0] * .4)
    block_high = int(grid_shape[0] * .5)
    for i in xrange(block_low, block_high):
        for j in xrange(block_low, block_high):
            grid[i][j] = 0.005

    # Evolve the initial conditions
    start = time.time()
    for i in range(num_iterations):
        grid = evolve(grid, 0.1)
    return time.time() - start
```

❶ 这里使用的初始条件跟图 6-2 的方形例子相同。

我们特地选择了足够小的 dt 和矩阵元素的值来让算法稳定。对这个算法的收敛特性更深入的讨论见 Willian Press 等人的著作《Numerical Recipes》第三版。

分配次数太多带来的问题

通过对纯 Python 演化函数使用 line_profiler，我们就能开始理解运行变慢可能是什么原因导致的。见例 6-5 的分析输出，我们看到函数大多数时间花在了求导和更新矩阵上。[①]这正是我们所需要的，因为这是一个纯粹的 CPU 密集型问题——任何不是花在 CPU 上的时间都是一个明显的优化对象。

例 6-5　纯 Python 2 阶扩散函数分析

```
$ kernprof.py -lv diffusion_python.py
Wrote profile results to diffusion_python.py.lprof
Timer unit: 1e-06 s

File: diffusion_python.py
```

① 这段受分析代码来自例 6-3，截取了部分以适应页面边界。别忘了 kernprof.py 需要函数被@profile 修饰才能对其进行分析（见 2.8 节）

```
Function: evolve at line 8
Total time: 16.1398 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     8                                           @profile
     9                                           def evolve(grid, dt, D=1.0):
    10        10           39      3.9      0.0      xmax, ymax = grid_shape # ❶
    11   2626570      2159628      0.8     13.4      new_grid = ...
    12      5130         4167      0.8      0.0      for i in xrange(xmax): # ❷
    13   2626560      2126592      0.8     13.2          for j in xrange(ymax):
    14   2621440      4259164      1.6     26.4              grid_xx = ...
    15   2621440      4196964      1.6     26.0              grid_yy = ...
    16   2621440      3393273      1.3     21.0              new_grid[i][j] = ...
    17        10           10      1.0      0.0         return grid # ❸
```

❶ 这条语句用了这么长时间是因为 grid_shape 必须从本地名字空间获得（见 4.2 节）。

❷ 这行命中了 5130 次，意味着我们操作的矩阵的 xmax 为 512。因为需要对 xrange 中的每一个值求值 512 次加上对循环终止条件求一次值，而这些一共发生了 10 次。

❸ 这行命中 10 次，告诉我们这个被分析函数运行了 10 次。

但是，输出也显示我们有 20%的时间花费在分配 new_grid 列表上。这是一个浪费，因为 new_grid 的属性不会发生变化——无论我们传入 evolve 的值是什么，new_grid 列表始终具有相同的形状和大小，包含同样的值。一个简单优化是只分配一次并重用它。这种优化类似于将重复代码移至循环外：

```
from math import sin

def loop_slow(num_iterations):
    """
    >>> %timeit loop_slow(int(1e4))
    100 loops, best of 3: 2.67 ms per loop
    """
    result = 0
    for i in xrange(num_iterations):
        result += i * sin(num_iterations) # ❶
    return result

def loop_fast(num_iterations):
    """
    >>> %timeit loop_fast(int(1e4))
    1000 loops, best of 3: 1.38 ms per loop
    """
    result = 0
```

```
    factor = sin(num_iterations)
    for i in xrange(num_iterations):
        result += i
    return result * factor
```

❶ sin(num_iterations)的值在循环内不会改变，所以不需要每次重新计算它。

我们可以对扩散代码进行类似的转换，如例 6-6 所示。在此，我们将例 6-4 的 new_grid 实例化之后再传入我们的 evolve 函数。该函数做的事情跟之前一样：读取 grid 列表并写入新的 new_grid 列表。然后，我们可以简单交换 new_grid 和 grid 并继续。

例 6-6　减少内存分配之后的纯 Python 2 阶扩散函数

```
def evolve(grid, dt, out, D=1.0):
    xmax, ymax = grid_shape
    for i in xrange(xmax):
        for j in xrange(ymax):
            grid_xx = grid[(i+1)%xmax][j] + grid[(i-1)%xmax][j] - 2.0 * grid[i][j]
            grid_yy = grid[i][(j+1)%ymax] + grid[i][(j-1)%ymax] - 2.0 * grid[i][j]
            out[i][j] = grid[i][j] + D * (grid_xx + grid_yy) * dt

def run_experiment(num_iterations):
    # 设置初始条件
    xmax,ymax = grid_shape
    next_grid = [[0.0,] * ymax for x in xrange(xmax)]
    grid = [[0.0,] * ymax for x in xrange(xmax)]

    block_low = int(grid_shape[0] * .4)
    block_high = int(grid_shape[0] * .5)
    for i in xrange(block_low, block_high):
        for j in xrange(block_low, block_high):
            grid[i][j] = 0.005

    start = time.time()
    for i in range(num_iterations):
        evolve(grid, 0.1, next_grid)
        grid, next_grid = next_grid, grid
    return time.time() - start
```

在例 6-7 的分析结果中[①]，我们可以看到经过这个细小的修改后的版本给我们带来了 21%的速度提升。这个结论跟之前对列表的 append 操作给我们的结论（见 3.2.1 节）类似：内存分配不便宜。每次当我们需要内存用于存储一个变量或列表，Python 都必须花时间向操作系统申请更多内存空间，然后还要遍历新分配的空间来将它初始化为某个值。任何时候只要有可能，重用已分配的内存空间都能为我们带来速度

① 这段受分析代码来自例 6-6，截取了部分以适应页面边界。

提升。不过，要小心实现这些改动。速度提升固然可观，你还是需要通过分析来确保这是你想要的结果，而不是污染你代码库的垃圾。

例 6-7　减少内存分配后的 Python 扩散函数的分析结果

```
$ kernprof.py -lv diffusion_python_memory.py
Wrote profile results to diffusion_python_memory.py.lprof
Timer unit: 1e-06 s

File: diffusion_python_memory.py
Function: evolve at line 8
Total time: 13.3209 s

Line #      Hits       Time  Per Hit  % Time  Line Contents
==============================================================
     8                                        @profile
     9                                        def evolve(grid, dt, out, D=1.0):
    10        10         15      1.5     0.0      xmax, ymax = grid_shape
    11      5130       3853      0.8     0.0      for i in xrange(xmax):
    12   2626560    1942976      0.7    14.6          for j in xrange(ymax):
    13   2621440    4059998      1.5    30.5              grid_xx = ...
    14   2621440    4038560      1.5    30.3              grid_yy = ...
    15   2621440    3275454      1.2    24.6              out[i][j] = ...
```

6.3　内存碎片

我们在例 6-6 写的 Python 代码还有一个问题，也是使用 Python 进行此类矢量计算的核心问题：原生 Python 并不支持矢量操作。这有两个原因：Python 列表存储的是指向实际数据的指针，且 Python 字节码并没有针对矢量操作进行优化，所以 for 循环无法预测何时使用矢量操作能带来好处。

Python 列表储存的是*指针*意味着列表保存的并不是实际的数据，而是这些数据的位置。在大多数情况下，这是一种优势，因为它允许我们在列表中存储任意类型的数据。但是对于矢量和矩阵操作，这会导致性能的大幅下降。

性能下降的原因在于每次我们需要获得 `grid` 矩阵中的元素时，我们都必须进行多次查找。比如，`grid[5][2]`这样的语句需要我们首先对列表 `grid` 查找索引 `5`。这会返回一个指针指向数据所在的位置。然后我们需要第二次列表查找找到索引 `2` 的元素。查到之后，我们得到的是实际数据所储存的位置。

一次这样的查找开销并不大，在大多数情况下可以忽略。但是，如果我们想要定位的数据在内存的一个连续存储块内，我们本来可以一次性将所有的数据全部读取出来，而不是分别对每个元素进行两次操作。这是数据分片的一个主要问题：当你的

数据被分成小片，你只能对每一片分别进行传输，而不是一次性传输整个块。这意味着你引入了更多的内存传输开销，且强制 CPU 在数据传输的过程中等待。我们可以用 perf 看到在缓存失效的情况下这个问题会有多严重。

这个在正确的时候将正确的数据传输给 CPU 的问题被称为“冯诺伊曼瓶颈”。意思是现代计算机所使用的层次化的内存架构会导致 CPU 和内存之间的带宽受到限制。如果我们数据传输的速度可以无限快，我们就不需要任何缓存，因为 CPU 可以立即获得任何它需要的数据。此时瓶颈就不再存在。

由于数据传输的速度不可能无限快，我们必须从 RAM 中预取数据并将其保存在一个更小但更快的 CPU 缓存中，并希望当 CPU 需要某个数据时，它可以从中更快读取到。虽然这已经是一个严重理想化了的场景，我们依然可以看到其中的一些问题——我们如何知道未来需要哪些数据？CPU 内部的分支预测和流水线技术会试图在处理当前指令的同时预测其下一条指令并将相应的内存读进缓存。但是减少瓶颈最好的方法是让代码知道如何分配我们的内存以及如何使用我们的数据进行计算。

探测内存移动至 CPU 的性能相当困难，不过，Linux 上的 perf 工具可以让我们洞察 CPU 如何处理运行中的程序。比如，我们可以对例 6-6 的纯 Python 代码运行 perf 并看到 CPU 运行我们代码的效率。结果见例 6-8。注意该例以及之后的 perf 例子的输出都被截取以适应页面边界。被删除的数据包括各测量值的方差，用于表示测量值在几次测量中发生了多大的变化。这有助于看到测量值在多大程度上依赖于实际的程序特性以及多大程度上受到来自系统的其他干扰，比如其他正在使用系统资源的程序。

例 6-8　减少内存分配后纯 Python 2 阶扩散函数的性能指标

```
$ perf stat -e cycles,stalled-cycles-frontend,stalled-cycles-backend,instructions,\
    cache-references,cache-misses,branches,branch-misses,task-clock,faults,\
    minor-faults,cs,migrations -r 3 python diffusion_python_memory.py

Performance counter stats for 'python diffusion_python_memory.py' (3 runs):

 329,155,359,015 cycles                    #    3.477 GHz
  76,800,457,550 stalled-cycles-frontend   #   23.33% frontend cycles idle
  46,556,100,820 stalled-cycles-backend    #   14.14% backend cycles idle
 598,135,111,009 instructions              #    1.82 insns per cycle
                                           #    0.13 stalled cycles per insn
      35,497,196 cache-references          #    0.375 M/sec
      10,716,972 cache-misses              #   30.191 % of all cache refs
133,881,241,254 branches                   # 1414.067 M/sec
   2,891,093,384 branch-misses             #    2.16% of all branches
    94678.127621 task-clock                #    0.999 CPUs utilized
```

```
            5,439 page-faults              #    0.057 K/sec
            5,439 minor-faults             #    0.057 K/sec
              125 context-switches         #    0.001 K/sec
                6 CPU-migrations           #    0.000 K/sec

     94.749389121 seconds time elapsed
```

6.3.1 理解 perf

让我们花一秒钟来理解 perf 告诉我们的各种性能指标以及它们跟我们代码的关系。task-clock 指标告诉我们的任务花了多少个时钟周期。这跟总体的运行时间不同，因为如果我们的程序花了一秒钟来运行但是使用了两个 CPU，那么 task-clock 将是 1000（task-clock 的单位是毫秒）。方便的是，perf 会帮我们计算并在该指标旁边告诉我们有多少个 CPU 被使用了。这个数字不完全等于 1 是因为进程有一段时间依赖于其他子系统的指令（比如分配内存时）。

context-switches 和 CPU-migrations 告诉我们程序在等待内核操作(如 I/O 操作）完成时，为了让其他进程得以运行，被挂起或迁移到另一个 CPU 核心上执行的次数。当一个 context-switch 发生时程序的执行会被挂起，让另一个程序得以执行。这是一个对时间要求非常精细的任务，也是我们需要尽量避免的，但是我们对它的发生无能为力。只要进程允许切换，内核就会接手；不过，我们可以做一些事来抑制内核切换我们的程序。总的来说，内核会在程序进行 I/O（比如读取内存、磁盘或网络）时将其挂起。我们在后续章节会看到，我们可以用异步操作来确保我们的程序在等待 I/O 时继续使用 CPU，这会让我们的进程继续运行而不被切换出去。另外，我们还可以设置程序的 nice 值来给我们的程序更高的优先级以防止内核将它切换出去。类似的，CPU-migrations 会发生在进程被挂起并迁移到另一个 CPU 上继续执行的情况，这是为了让所有的 CPU 都有同样程度的利用率。这可以被认为是一个特别糟糕的进程切换，因为我们的程序不仅被暂时挂起，而且丢失了 L1 缓存内所有的数据（每个 CPU 都有它自己的 L1 缓存）。

page-fault 是现代 UNIX 内存分配机制的一部分。分配内存时，内核除了告诉程序一个内存的引用地址以外没做任何事。但是，之后在这块内存第一次被使用时，操作系统会抛出一个缺页小中断，这将暂停程序的运行并正确分配内存。这被称为延迟分配系统。虽然这种手段相比以前的内存分配系统是一个很大的优化，缺页小中断本身依然是一个相当昂贵的操作，因为大多数操作都发生在你的程序外部。另外还有一种缺页大中断，发生于当你的程序需要从设备（磁盘、网络等）上请求还未被读取的数据时。这些操作更加昂贵，因为他们不仅中断了你的程序，还需要读取数据所在的设备。这种缺页不总是影响 CPU 密集的工作，但是，它会给任何需要读写磁盘或网络的程序带来痛苦。

一旦我们引用内存中的数据，数据就需要通过内存的多个层级（对此的讨论见 1.1.3 节“通信层”）。每当我们引用缓存中的数据，`cache-references` 指标就会增加。如果我们的缓存中还没有数据并需要从 RAM 获取，`cache-miss` 指标的计数就会增加。如果我们读取一个最近刚读过的数据（那个数据还在缓存里）或者读取的数据在最近刚读过的数据*附近*（缓存从 RAM 中按块读取数据），缓存失效就不会发生。缓存失效会拖慢 CPU 密集型工作，因为我们不仅需要等待数据从 RAM 被读取，而且中断了我们的执行流水线（马上会说到这个）。本章后面将讨论如何通过优化内存中的数据布局降低缓存失效。

`Instructions` 指标告诉我们的代码让 CPU 总共执行了多少条指令。流水线技术让 CPU 能在同一时间执行多条指令，`insns per cycle` 指标会告诉我们一个时钟周期内具体了几条。为了更好地优化流水线，`stalled-cycles-frontend` 和 `stalled-cycles-backend` 告诉我们的程序在等待流水线的前端或后端填满指令时等待了多少时钟周期。这可能是由于一次缓存失效，一次错误的分支预测或一次资源冲突。流水线前端负责从内存中获取下一条指令并解码成一个合法的操作，而后端则负责实际执行操作。有了流水线，CPU 就能在运行当前指令的同时获取并准备下一条指令。

`branch` 是代码执行流程发生改变的地方。想象一条 `if..then` 语句——基于条件语句的结果，我们会执行不同的代码段。这就是代码执行的分支——下一条指令将是两者之一。为了优化，特别是在使用了流水线的情况下，CPU 会试图猜测分支的走向并预取与其相关的指令。当预测出错时，就会发生 `stalled-cycles` 和 `branch-miss`。分支失效是相当令人困惑的，并可能导致很多奇怪的现象（比如，某些循环在排序列表上跑得远比乱序列表快，仅仅是因为发生了更少的分支失效）。

备忘

如果你想要一个在 CPU 层面各种性能指标更彻底的解释，请参考 Gurpur M. Prabhu 的“计算机架构导论”。它讲述在很底层发生的故事，让你对代码运行时底层发生的一些事有一个很好的理解。

6.3.2 根据 perf 输出做出抉择

例 6-8 的性能指标告诉我们在运行我们的代码时，CPU 需要访问 L1/L2 缓存 35 497 196 次，其中 10 716 972 次（30.191%）是由于请求的数据不在内存中而需要被获取。另外，我们可以看到每个 CPU 周期我们能平均执行 1.82 条指令，这告诉了我们通过流水线，乱序执行以及超线程技术（或者任何其他能够让你的 CPU 在一个时钟周期内执行多条指令的技术）给予我们的性能增幅。

数据的分片不仅增加了从内存传输数据到 CPU 的次数，还让你无法进行矢量计算，因为在计算时你的 CPU 缓存中并没有多个数据。我们在 1.1.3 节中解释过，矢量计算（或者说让 CPU 在同一时间进行多个计算）仅能发生在我们能够将相关数据填满 CPU 缓存的情况下。由于总线只能移动连续的内存数据，也就是说只有当格子中的数据在 RAM 中顺序储存时才有可能。由于我们的列表存储的是数据的指针而不是实际数据，格子中实际的值被分散在内存里，无法一次性全部复制。

我们可以通过使用 `array` 模块代替列表来减轻这个问题。`array` 对象可以在内存中顺序储存数据，一段 `array` 实际代表了一段连续的内存。但是这并没有完全解决问题——现在我们有了内存中的连续存储的数据，但 Python 依然不知道如何对我们的循环进行矢量计算。我们想要的是让原本一次只能处理一个数组元素的循环现在一次能够处理多个数据，但是之前说过，Python 没有这样的字节码优化（部分是由于 Python 语言极端的动态特性）。

> **备忘**
>
> 为什么我们顺序存储在内存中的数据不能自动矢量化？如果我们看看 CPU 运行的机器指令，就会发现矢量操作（比如两个数组相乘）和非矢量操作使用的是不同的 CPU 计算单元和指令集。为了让 Python 能够使用这些特殊指令，我们必须有一个模块专门用来使用这些指令集。我们马上会看到 `numpy` 如何让我们能够访问这些特殊指令。

另外，由于实现细节，使用 `array` 类型来创建数据列表实际上比使用 `list` 要慢。这是因为 `array` 对象存储的数据是一个非常底层的抽象，在用户使用它们时必须被转换成一个 Python 兼容的版本。这个额外的开销在你每次索引一个 `array` 类型时都会发生。这一实现细节导致了 `array` 对象不适用于数学计算而更适用于在内存中存放类型固定的数据。

6.3.3　使用 numpy

为了解决 `perf` 发现的分片问题，我们必须找到一个可以进行高效矢量操作的模块。幸运的是，`numpy` 拥有我们需要的所有特性——它能将数据连续存储在内存中并支持数据的矢量操作。结果就是，任何我们对 `numpy` 数组的数学操作都能自动矢量化而无须我们显式遍历每一个元素。这不仅仅让矩阵计算更简单，而且也更快。让我们看这样一个例子：

```
from array import array
import numpy
```

```
def norm_square_list(vector):
    """
    >>> vector = range(1000000)
    >>> %timeit norm_square_list(vector_list)
    1000 loops, best of 3: 1.16 ms per loop
    """
    norm = 0
    for v in vector:
        norm += v*v
    return norm

def norm_square_list_comprehension(vector):
    """
    >>> vector = range(1000000)
    >>> %timeit norm_square_list_comprehension(vector_list)
    1000 loops, best of 3: 913 µs per loop
    """
    return sum([v*v for v in vector])

def norm_squared_generator_comprehension(vector):
    """
    >>> vector = range(1000000)
    >>> %timeit norm_square_generator_comprehension(vector_list)
    1000 loops, best of 3: 747 µs per loop
    """
    return sum(v*v for v in vector)

def norm_square_array(vector):
    """
    >>> vector = array('l', range(1000000))
    >>> %timeit norm_square_array(vector_array)
    1000 loops, best of 3: 1.44 ms per loop
    """
    norm = 0
    for v in vector:
        norm += v*v
    return norm

def norm_square_numpy(vector):
    """
    >>> vector = numpy.arange(1000000)
    >>> %timeit norm_square_numpy(vector_numpy)
    10000 loops, best of 3: 30.9 µs per loop
    """
    return numpy.sum(vector * vector) # ❶

def norm_square_numpy_dot(vector):
    """
    >>> vector = numpy.arange(1000000)
```

```
    >>> %timeit norm_square_numpy_dot(vector_numpy)
    10000 loops, best of 3: 21.8 µs per loop
    """
    return numpy.dot(vector, vector) # ❷
```

❶ 这创建了 vector 上的两个隐式循环，一个用于做乘法，另一个求和。这两个循环类似 norm_squre_list_comprehension 中的循环，只是使用了 numpy 优化的数学代码。

❷ 这才是 numpy 中求矢量范数的推荐做法，通过矢量化的 numpy.dot 操作。前面提供的低效的 norm_squre_numpy 代码仅作说明用途。

这段简单的 numpy 代码运行速度是 norm_square_list 的 37.54 倍，比使用 Python 列表“优化”后的版本也要快 29.5 倍。纯 Python 循环和列表优化版本在速度上的差距体现了让后台进行更多计算相比于 Python 代码显式计算的优势。Python 是用 C 语言写的，利用 Python 内建机制进行运算，我们就获得了原生 C 代码的速度。我们在 numpy 代码中有如此大的速度提升也是基于同样的原因：我们使用了高度优化且特殊构建的对象取代了通用的列表结构来处理数组。

除了更轻量级且特化的机制，numpy 对象还给了我们内存的本地性和矢量操作，它们在处理数值计算时尤其重要。CPU 是超快的，大多数时候，仅仅提升其获取所需的数据的速度就是最好的优化手段。我们之前使用 perf 工具运行每个函数显示了使用 array 的版本和纯 Python 版本的函数花费了大约 1×10^{11} 条指令，而 numpy 版本花费了大约 3×10^{9} 条。另外，array 和纯 Python 版本大约有 80%缓存失效而 numpy 只有大约 55%。

我们的 norm_square_numpy 代码中，在运算 vector * vector 时，numpy 会为我们隐式调用一个循环。该循环和我们在其他例子中显式调用的一样：遍历 vector 内的每一项，乘以其自身。但是，由于我们让 numpy 来干这件事而不是自己用 Python 代码实现，numpy 就可以进行任何它想要的优化。numpy 在后台有极其优化的 C 代码来专门使用 CPU 的矢量操作。另外，numpy 数组在内存中是连续储存的底层数字类型，和（array 模块中的）array 对象有同样的空间需求。

另外还有一个额外好处是，numpy 支持我们将问题重新描述为一个点积。这让我们可以用一个单一操作来计算我们想要的值，而不是计算两个矢量的乘积并求和。在图 6-3 中可以看到，由于该函数的特化，norm_numpy_dot 的效率大大超越了其他所有函数，这也是因为我们不需要存储 vector * vector 的中间结果。

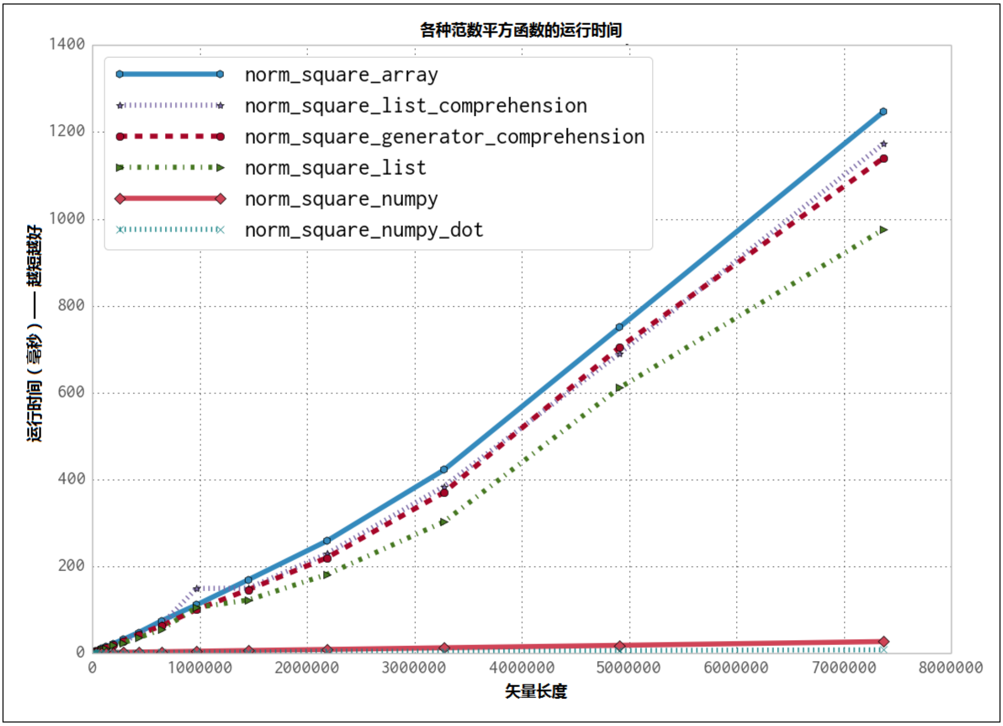


图 6-3 对不同长度的矢量求范数平方的函数运行时间

6.4 用 numpy 解决扩散问题

利用我们刚刚学到的 numpy，我们就可以轻松地把之前的纯 Python 代码矢量化。唯一需要引入的新功能是 numpy 的 roll 函数。该函数和我们手写的数组轮转技巧一样，但它运行在一个 numpy 数组上。一言以蔽之，它能将下面的数组轮转操作矢量化：

```
>>> import numpy as np
>>> np.roll([1,2,3,4], 1)
array([4, 1, 2, 3])

>>> np.roll([[1,2,3],[4,5,6]], 1, axis=1)
array([[3, 1, 2],
       [6, 4, 5]])
```

roll 函数创建了一个新的 numpy 数组，这一点有利有弊。坏处是我们需要额外的时间开销来分配新的空间，然后还需要用合适的数据填满它。另一方面，一旦我们创建了这个轮转后的新数组，我们立刻就能够在其上进行矢量操作，而不会受到 CPU 缓存失效的影响。这可以极大影响我们对矩阵数据运算的速度。在本章后续，

我们还将重写这部分代码来获得同样的好处而又不至于频繁分配更多内存。

有了这个额外的函数，我们就能将例 6-6 的 Python 扩散代码用更简单且矢量化的 numpy 数组重写。另外，我们将 grid_xx 和 grid_yy 的微分计算打散至另一个函数。例 6-9 展示了我们第一版的 numpy 扩散代码。

例 6-9　初版 numpy 扩散

```
import numpy as np

grid_shape = (1024, 1024)

def laplacian(grid):
    return np.roll(grid, +1, 0) + np.roll(grid, -1, 0) + \
           np.roll(grid, +1, 1) + np.roll(grid, -1, 1) - 4 * grid

def evolve(grid, dt, D=1):
    return grid + dt * D * laplacian(grid)

def run_experiment(num_iterations):
    grid = np.zeros(grid_shape)

    block_low = int(grid_shape[0] * .4)
    block_high = int(grid_shape[0] * .5)
    grid[block_low:block_high, block_low:block_high] = 0.005

    start = time.time()
    for i in range(num_iterations):
        grid = evolve(grid, 0.1)
    return time.time() - start
```

我们会立刻发现这段代码短多了。这通常预示着高性能：我们将很多重活累活移到 Python 解释器之外，让模块内建的优化代码来帮我们解决特定的问题（但是，这必须要经过测试！）。我们的假设是 numpy 有更好的内存管理可以更快地给 CPU 提供所需的数据。但是，这一点实际取决于 numpy 的实现，让我们对代码进行性能分析来验证我们的假设。例 6-10 显示了结果。

例 6-10　numpy 2 阶扩散的性能指标

```
$ perf stat -e cycles,stalled-cycles-frontend,stalled-cycles-backend,instructions,\
  cache-references,cache-misses,branches,branch-misses,task-clock,faults,\
  minor-faults,cs,migrations -r 3 python diffusion_numpy.py

Performance counter stats for 'python diffusion_numpy.py' (3 runs):
10,194,811,718 cycles                    #    3.332 GHz
 4,435,850,419 stalled-cycles-frontend #   43.51% frontend cycles idle
```

```
   2,055,861,567 stalled-cycles-backend  #   20.17% backend cycles idle
  15,165,151,844 instructions            #     1.49 insns per cycle
                                         #     0.29 stalled cycles per insn
     346,798,311 cache-references        #  113.362 M/sec
         519,793 cache-misses            #    0.150 % of all cache refs
   3,506,887,927 branches                # 1146.334 M/sec
       3,681,441 branch-misses           #    0.10% of all branches
     3059.219862 task-clock              #    0.999 CPUs utilized
         751,707 page-faults             #    0.246 M/sec
         751,707 minor-faults            #    0.246 M/sec
               8 context-switches        #    0.003 K/sec
               1 CPU-migrations          #    0.000 K/sec

     3.061883218 seconds time elapsed
```

这个结果显示出，相比于例 6-8 的降低内存分配的纯 Python 实现，仅仅改用 numpy 就给我们带来了 40 倍的速度提升。这是怎么做到的？

首先，我们要感谢 numpy 提供的矢量操作。虽然 numpy 版本看上去每个周期运行的指令较少，但是每条指令干了更多的活。也就是说，一条矢量操作指令可以对 4 个（或更多）数组元素进行乘法操作而不需要 4 条独立的乘法指令。最终导致我们解决同样问题的指令数变得更少。

numpy 版本的函数能够降低指令总数还有其他原因。其中之一是纯 Python 版本需要完整的 Python API 可用，而 numpy 版本则没有这样的需求——比如说，纯 Python 矩阵只能在 Python 中附加，而不能在 numpy 模块内被附加。即使我们并没有显式使用这个功能（以及还有其他很多功能），仅仅是为了让系统确保它能够被使用也会带来额外的开销。由于 numpy 可以假定它存储的数据永远是数字，所有跟数组相关的操作都可以针对数字操作优化。等我们讨论 Cython（7.6 节）时，我们会为了性能继续移除各种必要功能，它甚至可以移除列表边界检查来加速整个列表查询。

通常，指令的数量跟性能并不相关——拥有较少指令的程序可能并没有高效地运行它们，或者它们可能都是慢速指令。不过，我们看到除了降低指令数量以外，numpy 版本还减少了很多低效之处：缓存失效（仅 0.15%，相比于之前的 30.2%）。我们在 6.3 节中解释过，由于 CPU 必须等待从较慢的内存中读取数据而不是从缓存中直接使用，缓存失效降低了计算速度。事实上，内存碎片是如此决定性的性能因素，以至于在 numpy 中禁用矢量操作[①]并保持其他一切不变的情况下，我们仍旧能看到一个相对纯 Python 版本极大的速度提升（例 6-11）。

① 为此我们以-O0 开关编译 numpy。为了这个实验，我们用如下命令编译了 numpy1.8.0：
`$ OPT='-O0' FOPT='-O0' BLAS=None LAPACK=None ATLAS=None python setup.py build`。

例 6-11　禁用矢量操作的 numpy 2 阶扩散性能指标

```
$ perf stat -e cycles,stalled-cycles-frontend,stalled-cycles-backend,instructions,\
  cache-references,cache-misses,branches,branch-misses,task-clock,faults,\
  minor-faults,cs,migrations -r 3 python diffusion_numpy.py

Performance counter stats for 'python diffusion_numpy.py' (3 runs):

  48,923,515,604 cycles                     #    3.413 GHz
  24,901,979,501 stalled-cycles-frontend    #   50.90% frontend cycles idle
   6,585,982,510 stalled-cycles-backend     #   13.46% backend cycles idle
  53,208,756,117 instructions               #    1.09 insns per cycle
                                            #    0.47 stalled cycles per insn
      83,436,665 cache-references           #    5.821 M/sec
       1,211,229 cache-misses               #    1.452 % of all cache refs
   4,428,225,111 branches                   #  308.926 M/sec
       3,716,789 branch-misses              #    0.08% of all branches
    14334.244888 task-clock                 #    0.999 CPUs utilized
         751,185 page-faults                #    0.052 M/sec
         751,185 minor-faults               #    0.052 M/sec
              24 context-switches           #    0.002 K/sec
               5 CPU-migrations             #    0.000 K/sec

    14.345794896 seconds time elapsed
```

这个结果展示出：当我们使用 numpy 的时候，给我们带来 40 倍速度提升的决定性因素并不是矢量操作指令集，而是内存的本地性以及减少的内存碎片。事实上，从之前的实验上我们可以看到，矢量操作仅给我们带来了 40 倍速度提升中的 15%。[①]

内存问题才是代码效率低下的决定性因素这一认知并不会令我们感到太过震惊。计算机就是专门被设计用来处理我们的问题的——把数字相乘和相加。瓶颈就取决于能否将这些数字以足够快的速度传输给 CPU 让它能够以最高的速度进行计算。

6.4.1　内存分配和就地操作

为了优化内存方面的影响，让我们试着使用跟例 6-6 同样的方法来降低我们 numpy 代码的内存分配次数。内存分配比我们之前讨论的缓存失效更糟。不仅仅要在缓存中找不到数据时去 RAM 中找到正确的数据，内存分配还必须向操作系统请求一块可用的数据并保留它。向操作系统进行请求所需的开销比简单填充缓存大很多——填充一次缓存失效是一个在主板上优化过的硬件行为，而内存分配则需要跟另一个进程、内核打交道来完成。

为了移除例 6-9 中的内存分配，我们会在代码开头预分配一些空间然后只使用就地

① 这一点视实际使用的 CPU 而定。

操作。就地操作，比如+=、*=等，将其中一个输入重用为输出。这意味着我们不需要为计算的结果分配空间。

为了显式说明这点，我们看看当我们操作一个 numpy 数组时它的 id 如何改变（例 6-12）。id 是一个很好的追溯 numpy 数组的方式，因为 id 跟指向的内存区域有关。如果两个 numpy 数组具有相同的 id，那么他们指向同一块内存区域。[①]

例 6-12 就地操作减少内存分配

```
>>> import numpy as np
>>> array1 = np.random.random((10,10))
>>> array2 = np.random.random((10,10))
>>> id(array1)
140199765947424 ❶
>>> array1 += array2
>>> id(array1)
140199765947424 #❷
>>> array1 = array1 + array2
>>> id(array1)
140199765969792 #❸
```

❶❷ 这两个 id 相同，因为我们使用了就地操作。这意味着 array1 的内存地址没有改变，我们只是改变了其内部的数据。

❸ 内存地址在这里改变了。在计算 array1 + array2 时，一个新的内存地址被分配并填入计算的结果。不过这样做也有好处，比如当原始数据需要被保留时（比如，array3 = array1 + array2 允许你继续使用 array1 和 array2，而就地操作销毁了部分原始数据）。

另外，我们看到一个由非就地操作带来的期望中的性能损失。对于小的 numpy 数组，这一开销大约占了整个计算时间的 50%。至于大型计算，速度提升大概在几个百分点，但如果发生几百万次，这依然代表着一段很长的时间。在例 6-13 中，我们看到对小数组使用就地操作给我们带来了 20%的速度提升。这一数字会随着数组长度增加而变得更多，因为内存分配会变得更繁重。

例 6-13 使用就地操作减少内存分配

```
>>> %%timeit array1, array2 = np.random.random((10,10)), np.random.random((10,10))
#❶
... array1 = array1 + array2
...
```

① 这并不严格为真，因为两个 numpy 数组可以指向相同的内存区域但使用不同的步进信息来对同样的数据做出不同的表达。这样两个 numpy 数组会具有不同的 id。Numpy 数组的 id 结构有很多微妙之处，在本书讨论范围之外。

```
100000 loops, best of 3: 3.03 us per loop

>>> %%timeit array1, array2 = np.random.random((10,10)), np.random.random((10,10))
... array1 += array2
...
100000 loops, best of 3: 2.42 us per loop
```

❶ 注意我们使用了`%%timeit`而不是`%timeit`，这让我们可以指定用于设置环境的代码免于时间测量。

虽然改写我们例 6-9 中的代码并不十分复杂，但就地操作的坏处在于它确实会让代码有一点难以阅读。在例 6-14 中我们可以看到这一重构的结果。我们初始化`grid`和`next_grid`矩阵，然后重复交换两者。`grid`是当前系统的已知信息，在运行了`evolve`之后，`next_grid`包含了更新后的信息。

例 6-14　将大多数 numpy 操作改为就地操作

```
def laplacian(grid, out):
    np.copyto(out, grid)
    out *= -4
    out += np.roll(grid, +1, 0)
    out += np.roll(grid, -1, 0)
    out += np.roll(grid, +1, 1)
    out += np.roll(grid, -1, 1)

def evolve(grid, dt, out, D=1):
    laplacian(grid, out)
    out *= D * dt
    out += grid

def run_experiment(num_iterations):
    next_grid = np.zeros(grid_shape)
    grid = np.zeros(grid_shape)

    block_low = int(grid_shape[0] * .4)
    block_high = int(grid_shape[0] * .5)
    grid[block_low:block_high, block_low:block_high] = 0.005

    start = time.time()
    for i in range(num_iterations):
        evolve(grid, 0.1, next_grid)
        grid, next_grid = next_grid, grid # ❶
    return time.time() - start
```

❶ 因为`evolve`的输出被保存在输出矩阵`next_grid`，我们必须为下一次迭代交换这两个变量来让`grid`包含最新的信息。这一交换操作非常便宜，因为仅仅互换了数据的引用，而不是数据本身。

记住因为我们想要让每个操作都就地完成，所以每一个矢量操作都必须在同一行完成。这可以让一句简单的 `A = A * B + C` 变得相当费解。因为 Python 非常强调可读性，所以我们必须确保这些改动给我们带来的速度提升足以弥补这一点。

比较例 6-15 和例 6-10 的性能指标，我们看到移除不需要的内存分配给我们的代码带来了 29%的速度提升。部分是因为降低了缓存失效，但主要还是因为降低了内存缺页。

例 6-15　numpy 就地操作内存的性能指标

```
$ perf stat -e cycles,stalled-cycles-frontend,stalled-cycles-backend,instructions,\
  cache-references,cache-misses,branches,branch-misses,task-clock,faults,\
  minor-faults,cs,migrations -r 3 python diffusion_numpy_memory.py

Performance counter stats for 'python diffusion_numpy_memory.py' (3 runs):

   7,864,072,570 cycles                    #    3.330 GHz
   3,055,151,931 stalled-cycles-frontend   #   38.85% frontend cycles idle
   1,368,235,506 stalled-cycles-backend    #   17.40% backend cycles idle
  13,257,488,848 instructions              #     1.69 insns per cycle
                                           #     0.23 stalled cycles per insn
     239,195,407 cache-references          #  101.291 M/sec
       2,886,525 cache-misses              #    1.207 % of all cache refs
   3,166,506,861 branches                  # 1340.903 M/sec
       3,204,960 branch-misses             #    0.10% of all branches
     2361.473922 task-clock                #    0.999 CPUs utilized
           6,527 page-faults               #    0.003 M/sec
           6,527 minor-faults              #    0.003 M/sec
               6 context-switches          #    0.003 K/sec
               2 CPU-migrations            #    0.001 K/sec

     2.363727876 seconds time elapsed
```

6.4.2　选择优化点：找到需要被修正的地方

从例 6-14 的代码中，似乎可以看到我们解决了手头大部分的问题：我们使用 numpy 降低了 CPU 负担，我们降低了解决问题所必要的内存分配次数。但是，永远有更多的调查等待着我们。如果我们对代码做一个逐行性能分析（例 6-16），就会看到我们大部分的工作都在 laplacian 函数中完成。事实上 evolve 函数有 93%的时间花费在 laplacian 函数中。

例 6-16　逐行性能分析显示 laplacian 花费了太多时间

```
Wrote profile results to diffusion_numpy_memory.py.lprof
Timer unit: 1e-06 s
```

```
File: diffusion_numpy_memory.py
Function: laplacian at line 8
Total time: 3.67347 s

Line #      Hits         Time   Per Hit   % Time  Line Contents
==============================================================
     8                                           @profile
     9                                           def laplacian(grid, out):
    10       500       162009     324.0      4.4     np.copyto(out, grid)
    11       500       111044     222.1      3.0     out *= -4
    12       500       464810     929.6     12.7     out += np.roll(grid, +1, 0)
    13       500       432518     865.0     11.8     out += np.roll(grid, -1, 0)
    14       500      1261692    2523.4     34.3     out += np.roll(grid, +1, 1)
    15       500      124139     2482.8     33.8     out += np.roll(grid, -1, 1)

File: diffusion_numpy_memory.py
Function: evolve at line 17
Total time: 3.97768 s

Line #      Hits         Time   Per Hit   % Time  Line Contents
==============================================================
    17                                           @profile
    18                                           def evolve(grid, dt, out, D=1):
    19       500      3691674    7383.3     92.8    laplacian(grid, out)
    20       500       111687     223.4      2.8    out *= D * dt
    21       500       174320     348.6      4.4    out += grid
```

laplacian 函数如此慢的原因可能有很多。但是，这里我们主要考虑两个高级因素。首先，在调用 np.roll 时会为创建新的矢量分配内存（我们可以通过查看该函数的说明文档来验证这一点）。这意味着即使我们在之前的重构中移除了 7 处内存分配，仍然有 4 处分配是我们遗漏的。另外，np.roll 是一个非常通用的函数，有很多代码用来处理特殊情况。既然我们已经完全清楚我们需要做什么（那就是将每个维度的第一列的数据移到最后一列），那么我们就可以重写这个函数来消除大部分无用的代码。我们甚至可以将 np.roll 的代码逻辑和 add 操作合并，生成一个高度专用的 roll_add 函数以最少的内存分配次数和最精简的逻辑来完成我们的任务。

例 6-17 显示了这个重构的细节。我们需要的仅是创建新的 roll_add 函数并让 laplacian 使用它。因为 numpy 支持复杂索引（fancy indexing），实现这样一个函数只需要保证不搞乱索引就行。但是，之前说过，这样的代码性能虽好，可读性却不怎么样。

警告

注意我们除了彻底测试之外，还在 docstring 上花费了额外的维护工作。当你在做类似这样的优化时，维护代码的可读性是非常重要的，这些步骤是为了确保你的代码以你期望的方式工作，让未来的程序开发者能够维护你的代码并在出问题时清楚地知道你的代码做了些什么。

例 6-17　创建我们自己的轮转函数

```
import numpy as np

def roll_add(rollee, shift, axis, out):
    """
    Given a matrix, rollee, and an output matrix, out, this function will
    perform the calculation:

        >>> out += np.roll(rollee, shift, axis=axis)

    This is done with the following assumptions:
        * rollee is 2D
        * shift will only ever be +1 or -1
        * axis will only ever be 0 or 1 (also implied by the first assumption)

Using these assumptions, we are able to speed up this function by avoiding
extra machinery that numpy uses to generalize the `roll` function and also
by making this operation intrinsically in-place.
"""
if shift == 1 and axis == 0:
    out[1:, :] += rollee[:-1, :]
    out[0 , :] += rollee[-1, :]
elif shift == -1 and axis == 0:
    out[:-1, :] += rollee[1:, :]
    out[-1 , :] += rollee[0, :]
elif shift == 1 and axis == 1:
    out[:, 1:] += rollee[:, :-1]
    out[:, 0 ] += rollee[:, -1]
elif shift == -1 and axis == 1:
    out[:, :-1] += rollee[:, 1:]
    out[:, -1] += rollee[:, 0]

def test_roll_add():
    rollee = np.asarray([[1,2],[3,4]])
    for shift in (-1, +1):
        for axis in (0, 1):
            out = np.asarray([[6,3],[9,2]])
            expected_result = np.roll(rollee, shift, axis=axis) + out
            roll_add(rollee, shift, axis, out)
```

```
        assert np.all(expected_result == out)

def laplacian(grid, out):
    np.copyto(out, grid)
    out *= -4
    roll_add(grid, +1, 0, out)
    roll_add(grid, -1, 0, out)
    roll_add(grid, +1, 1, out)
    roll_add(grid, -1, 1, out)
```

如果我们查看例 6-18 中本次重写后的性能指标，我们会发现虽然它比例 6-14 有一个很大的性能提升（实际上提升了 70%），但是大多数性能指标都是相同的。`Page-faults` 的数量下降了，但是下降了不到 70%。同样，`cache-misses` 跟 `cache-references` 也下降了，但是也不足以导致整体这么大的性能提升。这里最重要的参数是 `instructions` 指标。`Instructions` 指标记录了 CPU 需要执行多少指令来完成整个程序——也就是说，CPU 需要干多少事情。使用特化的 `roll_add` 函数后 `instructions` 指标降低了 2.86 倍。这是因为我们不再需要依靠 `numpy` 提供的完整功能来轮转我们的矩阵，而是利用我们对数据的了解（我们的数据只有两个维度且只需轮转一次）创建了更精简的功能。我们将在 7.6 节中继续讨论关于在 `numpy` 和 Python 中精简不需要功能的主题。

例 6-18　numpy 使用就地内存操作和特化 laplacian 函数的性能指标

```
$ perf stat -e cycles,stalled-cycles-frontend,stalled-cycles-backend,instructions,\
    cache-references,cache-misses,branches,branch-misses,task-clock,faults,\
    minor-faults,cs,migrations -r 3 python diffusion_numpy_memory2.py

Performance counter stats for 'python diffusion_numpy_memory2.py' (3 runs):

   4,303,799,244 cycles                    #   3.108 GHz
   2,814,678,053 stalled-cycles-frontend   #  65.40% frontend cycles idle
   1,635,172,736 stalled-cycles-backend    #  37.99% backend cycles idle
   4,631,882,411 instructions              #    1.08 insns per cycle
                                           #    0.61 stalled cycles per insn
     272,151,957 cache-references          # 196.563 M/sec
       2,835,948 cache-misses              # 1.042 % of all cache refs
     621,565,054 branches                  # 448.928 M/sec
       2,905,879 branch-misses             #   0.47% of all branches
     1384.555494 task-clock                #   0.999 CPUs utilized
           5,559 page-faults               #   0.004 M/sec
           5,559 minor-faults              #   0.004 M/sec
               6 context-switches          #   0.004 K/sec
               3 CPU-migrations            #   0.002 K/sec
     1.386148918 seconds time elapsed
```

6.5 numexpr：让就地操作更快更简单

numpy 对矢量操作优化的一个缺陷是它一次只能处理一个操作。这意味着，当我们对 numpy 矢量进行 A * B + C 这样的操作时，首先要等待 A * B 操作完成，数据保存在一个临时矢量中，然后将这个新的矢量和 C 相加。正如例 6-14 中使用就地操作的扩散代码所示。

然而，有许多模块可以对这点进行优化。numexpr 模块可以将整个矢量表达式编译成非常高效的代码，可以将缓存失效以及临时变量的数量最小化。另外，它还能利用多个 CPU 核心（更多信息见第 9 章）以及 Intel 芯片专用的指令集来将速度最大化。

很容易修改代码来使用 numexpr：我们只需要将表达式重写为使用本地变量的字符串即可。表达式会在后台被编译成优化过的代码（并被缓存来确保相同的表达式不会导致同样的编译过程发生多次）并运行。例 6-19 显示了 evolve 函数改用 numexpr 是多么的简单。我们在 evaluate 函数中使用 out 参数，这样 numexpr 就不会为返回一个新矢量而分配内存。

例 6-19 使用 numexpr 来进一步优化大矩阵操作

```
from numexpr import evaluate

def evolve(grid, dt, next_grid, D=1):
    laplacian(grid, next_grid)
    evaluate("next_grid*D*dt+grid", out=next_grid)
```

numexpr 的一个重要特点是它考虑到了 CPU 缓存。它特地移动数据来让各级 CPU 缓存能够拥有正确的数据让缓存失效最小化。当我们对更新后的代码运行 perf（例 6-20）时，我们看到速度的确是提升了。但是，如果我们针对一个较小的 512×512 的矩阵（见本章最后的图 6-4）比较速度，我们会看到大约 15%的速度下降。这是为什么？

例 6-20 使用了 numpy 就地内存操作特化 laplacian 函数以及 numexpr 的性能指标

```
$ perf stat -e cycles,stalled-cycles-frontend,stalled-cycles-backend,instructions,\
  cache-references,cache-misses,branches,branch-misses,task-clock,faults,\
  minor-faults,cs,migrations -r 3 python diffusion_numpy_memory2_numexpr.py

Performance counter stats for 'python diffusion_numpy_memory2_numexpr.py' (3 runs):

   5,940,414,581 cycles                    #   1.447 GHz
   3,706,635,857 stalled-cycles-frontend   #  62.40% frontend cycles idle
```

```
      2,321,606,960 stalled-cycles-backend   #  39.08% backend cycles idle
      6,909,546,082 instructions             #    1.16 insns per cycle
                                             #    0.54 stalled cycles per insn
        261,136,786 cache-references         #  63.628 M/sec
         11,623,783 cache-misses             #   4.451 % of all cache refs
        627,319,686 branches                 # 152.851 M/sec
          8,443,876 branch-misses            #   1.35% of all branches
        4104.127507 task-clock               #   1.364 CPUs utilized
              9,786 page-faults              #   0.002 M/sec
              9,786 minor-faults             #   0.002 M/sec
              8,701 context-switches         #   0.002 M/sec
                 60 CPU-migrations           #   0.015 K/sec

        3.009811418 seconds time elapsed
```

numexpr 引入的大多数额外的机制都跟缓存相关。当我们的矩阵较小且计算所需的所有数据都能被放入缓存时，这些额外的机制只是白白增加了更多的指令而不能对性能有所帮助。另外，将字符串编译成矢量操作也会有很大的开销。当程序运行的整体时间较少时，这个开销就会变得相当引人注意。不过，当我们增加矩阵的大小时，我们会发现 numexpr 比原生 numpy 更好地利用了我们的缓存。而且，numexpr 利用了多核来进行计算并尝试填满每个核心的缓存。当矩阵较小时，管理多核的开销盖过了任何可能的性能提升。

我们用来跑测试的电脑有 20480 KB 的缓存（Intel Xeon E5-2680）。因为有两个数组需要处理，一个作为输入，另一个作为输出，所以我们可以轻易计算出需要足以填满缓存的矩阵大小。矩阵元素的总数是 20480 KB/64 bit = 2560000。因为我们有两个矩阵，这一总数被平分到两个对象中（所以每个对象最多可以有 2560000 / 2 = 1280000 个元素）。最后，对这个数字求平方根可以让我们知道能存放这么多元素的矩阵的大小。总的来说，这意味着大概两个大小为 1131 的 2 维数组就会填满缓存（$\sqrt{20480\text{KB}/64\text{bit}/2}=1131$）。但实际上，我们自己没办法完全填满缓存（其他程序会占用部分缓存），所以现实来说大概能填入两个 800 × 800 的数组。见表 6-1 和表 6-2，我们可以看到当矩阵大小从 512 × 512 跳到 1024 × 1024 时，numexpr 代码的性能就开始超越纯 numpy。

6.6 告诫故事：验证你的“优化”（scipy）

重要的是记住我们在本章中对每一次优化使用的一套方法：首先对代码进行性能分析来感知问题可能出在哪，提出一个可能的解决方案来修改慢的那部分，然后再次进行性能分析来确保我们的修改可行。虽然这听上去十分简单直接，但事情很快会变得复杂，比如我们看到 numexpr 的性能是如何受到矩阵大小影响的。

当然我们提出的方案并不总是可行。在为本章写代码时，有一次作者看到 laplacian 函数最慢并假设 scipy 的函数会明显更快。这一想法来自于一个事实，那就是 laplacian 在图像分析这一行是一个常见操作，可能已经有一个非常优化的库加速了其运行的速度。scipy 正好有一个图像子模块，我们一定会得到幸运的眷顾！

用 scipy 实现起来很简单（例 6-21）且不需要考虑实现周期边界的复杂度（因为 scipy 的 wrap 模式会帮我们搞定）。

例 6-21 使用 scipy 的 laplace 滤波

```
from scipy.ndimage.filters import laplace

def laplacian(grid, out):
    laplace(grid, out, mode='wrap')
```

能够简单实现这一点相当重要，且绝对为这个函数赢得了一些分数，在我们考查性能之前。然而一旦我们对 scipy 代码进行性能对比（例 6-22），我们才发觉：这个函数跟之前的相比（例 6-14）并没有带来大量的速度提升。事实上，当我们增加矩阵大小时，这个函数甚至开始变得性能下降了（见本章最后的图 6-4）。

例 6-22 使用 scipy 的 laplacian 函数的扩散代码性能指标

```
$ perf stat -e cycles,stalled-cycles-frontend,stalled-cycles-backend,instructions,\
  cache-references,cache-misses,branches,branch-misses,task-clock,faults,\
  minor-faults,cs,migrations -r 3 python diffusion_scipy.py

Performance counter stats for 'python diffusion_scipy.py' (3 runs):

   6,573,168,470 cycles                    #   2.929 GHz
   3,574,258,872 stalled-cycles-frontend   #  54.38% frontend cycles idle
   2,357,614,687 stalled-cycles-backend    #  35.87% backend cycles idle
   9,850,025,585 instructions              #    1.50 insns per cycle
                                           #    0.36 stalled cycles per insn
     415,930,123 cache-references          # 185.361 M/sec
       3,188,390 cache-misses              #   0.767 % of all cache refs
   1,608,887,891 branches                  # 717.006 M/sec
       4,017,205 branch-misses             #   0.25% of all branches
     2243.897843 task-clock                #   0.994 CPUs utilized
           7,319 page-faults               #   0.003 M/sec
           7,319 minor-faults              #   0.003 M/sec
              12 context-switche           #   0.005 K/sec
               1 CPU-migrations            #   0.000 K/sec

     2.258396667 seconds time elapsed
```

对比 scipy 版本和我们自己的 laplacian 函数性能指标（例 6-18），我们可以得到一些提示，为什么没有从这次重写中得到期待的性能提升。

指标中最突出的是 page-faults 和 instructions。scipy 版本的指标中两者的值都明显更大。page-faults 的增加显示出 scipy 的 laplace 函数虽然支持就地操作，但它依然分配了很多内存。事实上 scipy 版本的 page-faults 的次数比我们第一次用 numpy 重写的版本还要多（例 6-15）。

但更严重的还是 instructions 指标。它告诉我们 scipy 代码比我们的 laplacian 函数让 CPU 多做了超过两倍的工作。即使这些指令都是更加优化的（因为我们可以看到更高的 insns per cycle 计数，也就是 CPU 在一个时钟周期内能完成多少条指令），额外的优化并没有能胜过指令数的猛增。这可能部分是因为 scipy 代码写得非常通用，以使它可以处理具有不同边界条件的各种输入（所以需要额外的代码也就是更多的指令数）。事实上这一点我们也可以通过 scipy 代码拥有更多的 branches 指标看出。

6.7 小结

回顾我们的优化历程，看起来我们使用了两种方法：减少 CPU 获得数据的时间和减少 CPU 需要干的工作。表 6-1 和表 6-2 在不同的矩阵大小上显示了我们在原始的纯 Python 实现之上进行的各种优化努力后的结果对比。

图 6-4 显示了这些优化手段之间的比较。我们可以看到两种优化手段的三个基带：底部的基带显示了我们纯 Python 实现在进行了降低内存分配次数之后的小小提升，中间的基带显示了我们使用 numpy 并进一步减少内存分配后发生了什么，最上面的基带则显示了减少进程总体工作量的结果。

表 6-1　各种优化，各种矩阵大小，运行 500 次 evolve 函数的总时间

Method	256 × 256	512 × 512	1 024 × 1 024	2 048 × 2 048	4 096 × 4 096
Python	2.32s	9.49s	39.00s	155.02s	617.35s
Python + memory	2.56s	10.26s	40.87s	162.88s	650.26s
numpy	0.07s	0.28s	1.61s	11.28s	45.47s
numpy + memory	0.05s	0.22s	1.05s	6.95s	28.14s
numpy + memory + laplacian	0.03s	0.12s	0.53s	2.68s	10.57s
numpy + memory + laplacian + numexpr	0.04s	0.13s	0.50s	2.42s	9.54s
numpy + memory + scipy	0.05s	0.19s	1.22s	6.06s	30.31s

表 6-2　各种优化，各种矩阵大小，运行 500 次 evolve 函数相比原生 Python（例 6-3）的速度提升倍数

Method	256 × 256	512 × 512	1 024 × 1 024	2 048 × 2 048	4 096 × 4 096
Python	0.00x	0.00x	0.00x	0.00x	0.00x
Python + memory	0.90x	0.93x	0.95x	0.95x	0.95x
numpy	32.33x	33.56x	24.25x	13.74x	13.58x
numpy + memory	42.63x	42.75x	37.13x	22.30x	21.94x
numpy + memory + laplacian	77.98x	78.91x	73.90x	57.90x	58.43x
numpy + memory+ laplacian +numexpr	65.01x	74.27x	78.27x	64.18x	64.75x
numpy + memory + scipy	42.43x	51.28x	32.09x	25.58x	20.37x

我们从中学到的一个重要的教训是你应该总是将代码需要的任何管理性工作放在初始化阶段进行。这可能包括内存分配，读取配置文件，预先计算程序所需的一些数据等。原因有两点。首先在初始化阶段一次性搞定可以让你减少这些工作运行的总次数，并让你知道你可以在将来不需要付出什么代价就使用这些资源。其次，你的程序不会因为要转而去做这些工作而打扰了流程，这可以让流水线更有效并让缓存始终含有相关数据。

我们同时还学到了数据的本地性以及将数据传给 CPU 这一简单操作的重要性有多高。CPU 的缓存可以相当复杂，所以大多数时候我们都会让各种优化过的专用函数来处理它。但是，只要我们理解背后发生的故事并且用尽了各种可能的方法去优化内存的使用方法，那么结果会大不一样。比如，理解了缓存的工作方式，我们就能理解图 6-4 中的每一种优化手段在矩阵大到一定程度之后性能提升程度就不再发生变化的原因是我们的矩阵已经填满了 L3 缓存。建立内存分级制度本来是为了解决冯诺依曼瓶颈，但是当这种情况发生时，它反而成为了我们的制约。

另一个重要的收获是考虑使用各种外部库。Python 本身是一门非常容易使用的高可读性语言，让你能够快速地编写和调试代码。但是使用外部库对于优化性能来说是必要的。这些外部库可以超级快，因为它们都是用各种低级语言写的——但是因为它们提供了 Python 接口，你依然可以迅速写出使用它们的代码。

最后，我们学到了运行性能测试并对结果做出假设的重要性。在测试之前先进行假设，我们就能让结果告诉我们优化是否真的成功了。这个改动是否能加速运行时间？是否减少了内存分配？是否降低了缓存失效的数量？现代计算机系统的复杂性让优化变成了一门艺术，而能对性能指标进行定量分析则是一个很大的帮助。

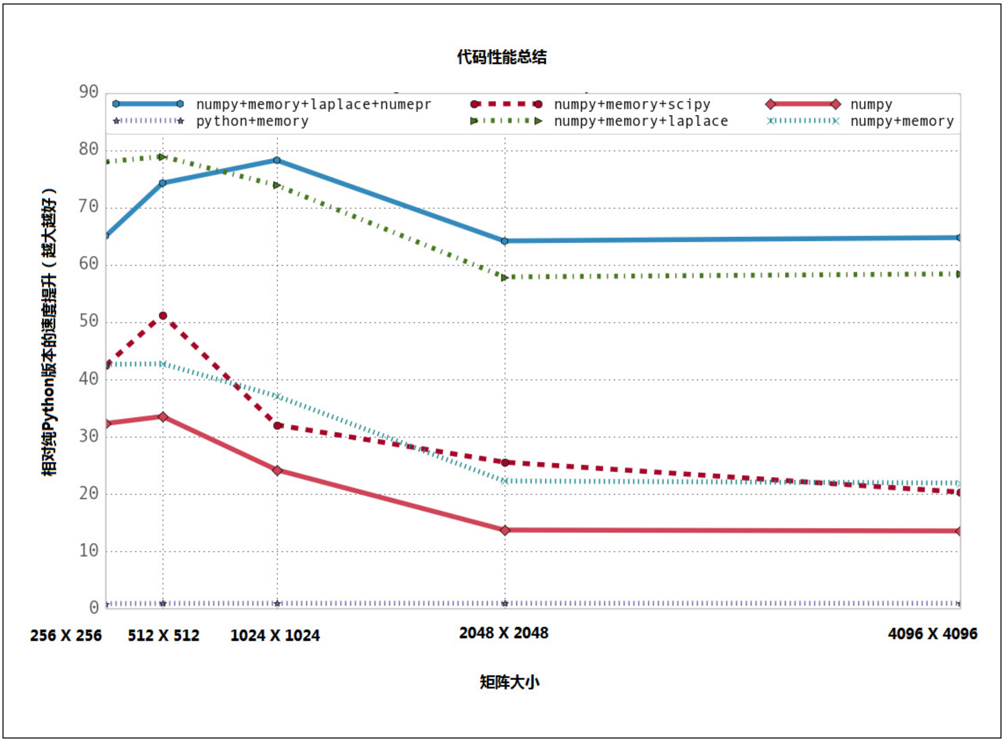


图 6-4　本章各种优化方法带来的速度提升总结

优化的最后一点是必须花很大精力确保优化在各种计算机上都是通用的（你的假设以及测试结果可能跟运行程序的计算机架构以及模块编译方式等相关）。另外，在进行这些优化时还必须考虑到其他开发者，你的改动会多大程度上影响代码的可读性。比如，我们意识到例 6-17 中实现的解决方案具有一定的模糊性，所以特地确保代码具有完备的描述文档和测试来帮助我们以及团队中的其他成员。

下一章，我们会谈到如何创建你自己的外部模块来更高效地解决特定的问题。这让我们能够用快速原型的方式来写程序——首先用较慢的代码解决问题，然后找到导致慢的原因，最终让它们快起来。经常进行性能分析来找到并优化我们的慢速代码段，就能让自己的程序运行尽可能快的同时还省了自己的时间。

第 7 章

编译成 C

读完本章之后你将能够回答下列问题

- 我怎样让我的 Python 代码作为低级代码来运行?
- JIT 编译器和 AOT 编译器的区别是什么?
- 编译后的 Python 代码运行什么任务能够比本地 Python 快?
- 为什么类型注解提升了编译后 Python 代码的运行速度?
- 我该怎样使用 C 或 Fortran 为 Python 编写模块?
- 我该怎样在 Python 中使用 C 或者 Fortran 的库?

让你的代码运行更快的最简单的办法就是让它做更少的工作。设想你已经选择了优秀的算法并且已经减少了要处理的数据量,要执行更少指令的最简单的方法就是把你的代码编译成机器码。

Python 为此提供了许多选项,包括纯粹的基于 C 的编译方式,比如 Cython、Shed Skin 和 Pythran,凭借 Numba 的基于 LLVM 的编译方式,还有替代虚拟机的 PyPy,包含了一个内置的即时编译器(JIT)。当决定采用哪条路线时,你需要在代码的可适用性和团队效率的要求方面做出权衡。

这些工具中的每一种都给你的工具链增加了新的依赖性,并且 Cython 会额外要求你用一种新的语言类型来编写(一种 Python 和 C 的混合),这就意味着你需要新的技能。Cython 的新语言可能会伤害你的团队效率,因为没有 C 语言知识的团队成员在支持这种代码方面可能会有麻烦,尽管在实践中,这很可能只是一个小小的顾虑,因为你只会在代码中精心选择的小部分区域使用 Cython。

值得注意的是对你的代码执行 CPU 和内存剖析有可能会促发你去思考可以采用的高层算法优化。这些算法的变动（例如，用额外的逻辑来避免计算或者用缓存来避免重新计算）会帮助你避免在代码中做无用功，并且 Python 的表达力会帮助你获得发现这些算法的机会。Radim Řehůřek 在 12.2 节中讨论了 Python 的实现怎样能够胜出纯 C 的实现。

在本章中，我们会检视：

- Cython ——这是编译成 C 最通用的工具，覆盖了 numpy 和普通的 Python 代码（需要一些 C 语言的知识）。
- Shed Skin —— 一个用于非 numpy 代码的，自动把 Python 转换成 C 的转换器。
- Numba —— 一个专用于 numpy 代码的新编译器。
- Pythran —— 一个用于 numpy 和非 numpy 代码的新编译器。
- PyPy —— 一个用于非 numpy 代码的，取代常规 Python 可执行程序的稳定的即时编译器。

接下来在本章中，我们会查看外来接口，这些接口允许 C 代码被编译进 Python 的扩展模块。Python 的本地 API 和 ctypes、cffi（来自 PyPy 的作者），以及 f2py 这种能把 Fortran 转为 Python 的转换器一起使用。

7.1 可能获得哪种类型的速度提升

如果你的问题求助于编译方式，那么很有可能得到至少一个数量级大小的速度提升。这里，我们会看到在单核上，以及在使用 OpenMP 的多核上，有各种各样的方法来达成一到两个数量级大小的提速。在编译后趋于更快运行的 Python 代码有可能是数学方面的，并且可能有许多循环在重复着多次相同的运算。在这些循环中，有可能会生成许多临时对象。

调用外部库（例如，正则表达式、字符串操作、调用数据库）的代码在编译后不可能表现出任何速度提升。I/O 密集型的程序同样不可能表现出明显的速度提升。

类似地，如果你的 Python 代码集中于调用向量化的 numpy 例程，那么在编译后就不大可能运行得更快——只有当被编译的代码主要是 Python（并且可能主要是循环）时才会运行得更快。我们在第 6 章会看到 numpy 运算，编译不会真正有助于提速，因为没有许多中间对象。

总体而言，编译后的代码不可能比手工精心编写的 C 例程运行得更快，但也不可能比它慢很多。从你的 Python 代码生成的 C 代码很有可能和手写的 C 例程跑得一样快，除非 C 程序员掌握了特定的知识和方法在目标机器架构上去调制 C 代码。

对于集中于数学方面的代码来说，一个手写的 Fortran 例程有可能会超越等价的 C 例程，但是这也有可能需要具备专家级别的知识水准。总体而言，一个编译后的结果（可能使用了 Cython、Pythran 或 Shed Skin）将会如大多数程序员所需要的那样接近于手写 C 的结果。

当你剖析和工作于你的算法时，请把图 7-1 记在脑中。通过少量剖析去理解你的代码的工作应该能够让你在算法层面做出更明智的选择。在这之后，致力于编译器使用上的一些工作应该让你获得额外的速度提升。你还可能会一直微调你的算法，但是不要惊讶于见到你那部分不断增加的工作量只是换来了越来越小的改进。要知道多余的努力可能是无效的。

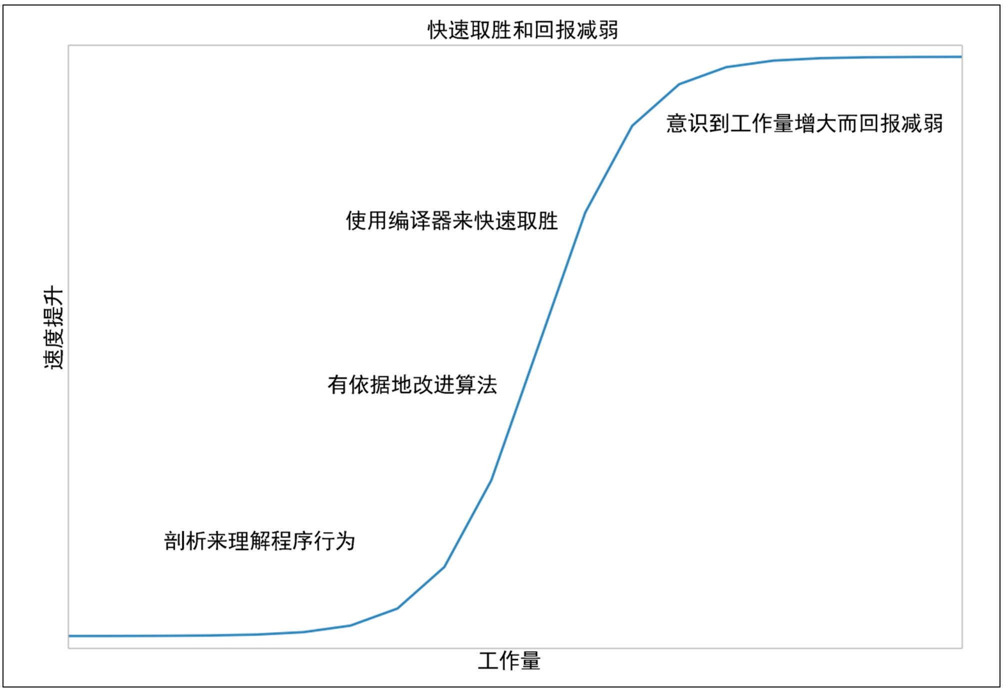


图 7-1　一些花在剖析和编译上的工作会带来丰厚回报，但是继续努力，收益就趋向于不断减少

如果你正在处理 Python 代码和内置的库，不涉及 numpy，那么 Cython、Shed Skin 和 PyPy 是你的主要选择。如果你正在用 numpy 工作，那么 Cython、Numba 和 Pythran 是正确的选择。这些工具都支持 Python2.7，一些支持 Python3.2+。

下面一些例子仅需要懂一点 C 编译器和 C 代码。如果你缺乏这方面知识，你应该在深入探索之前学习一点 C 语言和编译一个可以工作的 C 程序。

7.2 JIT 和 AOT 编译器的对比

我们将要查看的工具大体分为两类：提前编译工具（Cython、Shed Skin、Pythran）和“即时”编译工具（Numba、PyPy）。

通过提前编译（AOT），你会创建一个为你的机器定制的静态库。如果你下载了 `numpy`、`scipy` 或 `scikit-learn`，它就会在你的机器上用 Cython 编译部分的库（或者如果你正在使用像 Continuum’s Anaconda 之类的分发包，你就会使用一个事先构建的预编译库）。通过在使用之前编译的方式，你就会获得一个能够在工作中立即拿来使用来解决你的问题的库。

通过即时编译，你不必提前做很多（如果有的话），你让编译器在使用时只逐步编译恰到好处的那部分代码。这就意味着你会有“冷启动”问题——如果你的大部分代码能够被编译并且当前都还没有被编译过，当你开始运行代码而且正在被编译时，就会跑得很慢。如果这事在你每次运行脚本时都发生，并且你运行这脚本很多次，开销就会变得很显著。PyPy 会遭受这个问题，所以你可能不想要用它来处理短小且频繁运行的脚本。

当前情形向我们展示了提前编译可以给我们带来最快的速度提升，但这经常会需要大量的人力。即时编译提供了印象深刻的速度提升而且几乎不需要人工干预，但是它也会遇到刚才描述到的问题。当为你的问题选择正确的技术时，你不得不思考这些权衡问题。

7.3 为什么类型检查有助代码更快运行

Python 是动态类型的——一个变量能够引用任何类型的对象，并且任意代码行都能够改变被引用对象的类型。这使得虚拟机难以在机器码层面优化代码的运行方式，因为它不知道哪种基础数据类型会用于将来的运算。让代码保持泛型就会让代码运行更慢。

在下面的例子中，`v` 或是一个浮点数，或是一对代表复数的浮点数。两个条件会先后在相同循环的不同位置发生，或者在相关代码的串行区域发生：

```
v = -1.0
print type(v), abs(v)
```

```
<type 'float'> 1.0

v = 1-1j
print type(v), abs(v)

<type 'complex'> 1.41421356237
```

abs 函数会根据底层数据类型来以不同方式工作。对一个整数或浮点数来说，abs 只是简单地把负值转换为正值来作为结果。对复数来说，abs 涉及对分量的平方和开平方根：

$$abs(c) = \sqrt{c.real^2 + c.imag^2}$$

还是复数的例子，它的机器码涉及更多的指令并且会运行得更久。在对一个变量调用 abs 之前，Python 首先不得不查看变量的类型，接着决定调用哪个函数版本——当你做出很多重复调用的时候，这个开销会累积。

在 Python 的每种基础对象内部，就像 integer 那样，会被更高层的 Python 对象包装起来（例如，一个 int 包装了 integer）。更高层的对象有额外的函数，就像为辅助存储的__hash__函数，还有为打印的__str__函数。

在 CPU 密集型的代码区域内部，不改变变量类型的情况很常见。这就给了我们一个机会来做静态编译和加快代码运行。

如果我们想要的一切全都是大量的中间级的数学运算，我们就不需要更高层次的函数，我们也不大可能需要引用计数的机制。我们会向下进入到机器代码的层面，并且使用机器码和字节来快速运算，而不是去操控更高层次的 Python 对象，那会涉及更大的开销。要做到这个，我们就要提前决定对象的类型，这样我们就能产生正确的 C 代码。

7.4 使用 C 编译器

在接下来的示例中，我们会从 GNU C 编译器的工具链中使用 gcc 和 g++。如果你能正确配置环境，你可能会选择替代 gcc 的编译器（例如，Intel 的 icc 或者微软的 cl）。Cython 使用了 gcc，Shed Skin 使用了 g++。

gcc 对大多数平台来说是很好的选择，它得到了很好的支持还很先进。虽然经常有可能利用调制的编译器（例如 Intel 的 icc 可能会在 Intel 设备上生成比 gcc 更快的代码）来压榨出更高的性能，但是代价就是你不得不去获取一些更多的领域知识并且学习在替代 gcc 的编译器上怎样去调制开关项。

C 和 C++经常被用作静态编译器，而不用其他语言，比如 Fortran，这要归因于 C 和 C++的普适性和大范围的支持库。编译器和转换器（Cython 等，在这种情况下是转换器）有机会来学习注解代码从而决定静态优化步骤（就像内联函数和循环展开）是否要采用。对中间级的抽象语法树的贪婪分析（由 Pythran、Numba 和 PyPy 执行）提供了机会来结合 Python 表达法的知识，从而去通知底层编译器尽可能利用已经看见的模式。

7.5 复习 Julia 集的例子

回到第 2 章我们剖析了 Julia 集产生器。这个代码使用整数和复数来生成输出图片。图片的计算是 CPU 密集型的。在代码中主要的开销就是有个计算输出列表的内循环，这是 CPU 密集型的本质。这个列表能被当作方形的像素阵列来画出，其中每个值代表了产生那个像素的开销。

内循环的代码显示在例 7-1 中。

例 7-1 复习 Julia 函数的 CPU 密集型代码

```
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

在 Ian 的笔记本电脑上，原始的 Julia 集在 1000*1000 的格子中计算并且 `maxiter=300`，使用运行在 CPython2.7 中的纯粹 Python 实现，计算大概接近于 11 秒。

7.6 Cython

Cython 是一个能把类型注解的 Python 转换为一个扩展编译模块的编译器。类型注解如同 C 一样。能够像一个常规 Python 模块那样使用 import 来导入扩展模块。一开始简单，但是它有一个学习曲线，必须去攀登每一级更高的复杂度和优化度。对 Ian 来说，Cython 是一个可以选择的工具，用来把计算密集型的函数转为更快的代码，这是由于它的广泛使用性、成熟性和对 OpenMP 的支持。

随着 OpenMP 标准，它可能把并行问题转换为在一台多核 CPU 机器上运行的多处理器感知模块。线程从你的 Python 代码隐藏起来了，它们通过生成的 C 代码来运算。

Cython（2007 年发布）是 Pyrex（2002 年发布）的一个分支，在原始 Pyrex 基础上扩展了能力。使用 Cython 的库包括 `scipy`、`scikit-learn`、`lxml` 和 `zmp`。

Cython 能够被用来通过一个 setup.py 脚本编译一个模块。它也能够在 IPython 中通过一个“magic”命令来交互使用。通常情况下类型由开发者注解，尽管一些自动注解也是有可能的。

7.6.1 使用 Cython 编译纯 Python 版本

开始写一个扩展编译模块的简单方法涉及 3 个文件。使用我们的 Julia 作为例子，它们是：

- 调用它的 Python 代码（来自前面的 Julia 代码块）。
- 在一个新.pyx 文件中要被编译的函数。
- 一个 setup.py，包含了调用 Cython 来制作扩展模块的指令。

使用这个方法，setup.py 脚本调用 Cython 把.pyx 文件编译成一个编译模块。在类 UNIX 系统上，编译模块可能会是一个.so 文件；在 Windows 上应该是一个.pyd（类 DLL 的 Python 库）。

对于 Julia 的例子，我们会用：

- julia1.py，构建输入列表并调用计算函数。
- cythonfn.pyx，包含了我们能注解的 CPU 密集型函数。
- setup.py，包含了构建指令。

运行 setup.py 的结果就是获得一个能够被导入的模块。在例 7-2 的 julia1.py 脚本中，我们只需要做一些很小的改动来导入新的模块并调用我们的函数。

例 7-2　导入新编译模块到我们的主代码中

```
...
import calculate # as defined in setup.py
...
def calc_pure_python(desired_width, max_iterations):
    # ...
```

```
    start_time = time.time()
    output = calculate.calculate_z(max_iterations, zs, cs)
    end_time = time.time()
    secs = end_time - start_time
    print "Took", secs, "seconds"
...
```

在例 7-3 中，我们将从一个没有类型注解的纯 Python 脚本开始。

例 7-3　在 cythonfn.pyx（从.py 重命名）中为 Cython 的 setup.py 准备的未改动过的纯 Python 代码

```
# cythonfn.pyx
def calculate_z(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

在例 7-4 中显示的 setup.py 脚本是简短的，它定义了把 cythonfn.pyx 转换为 calculate.so 的方法。

例 7-4　setup.py，把 cythonfn.pyx 转换为由 Cython 去编译的 C 代码

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
        cmdclass = {'build_ext': build_ext},
        ext_modules = [Extension("calculate", ["cythonfn.pyx"])]
        )
```

当我们在例 7-5 中用参数 build_ext 运行 setup.py 脚本时，Cython 会查找 cythonfn.pyx 并构建 calculate.so。

备忘

记住这是一个手动步骤——如果你更新了你的.pyx 或者 setup.py，但是忘记重新去运行构建命令，你不会获得一个要导入的更新的.so 模块。如果你不确定是否编译了代码，检查.so 文件的时间戳。如果怀疑的话，删除生成的 C 文件和.so 文件，再重新构建。

例 7-5　运行 setup.py 构建一个新的编译模块

```
$ python setup.py build_ext --inplace
running build_ext
cythoning cythonfn.pyx to cythonfn.c
building 'calculate' extension
gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall
    -Wstrict-prototypes -fPIC -I/usr/include/python2.7 -c cythonfn.c
    -o build/temp.linux-x86_64-2.7/cythonfn.o
gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions -Wl,
    -Bsymbolic-functions -Wl,-z,
    relro build/temp.linux-x86_64-2.7/cythonfn.o -o calculate.so
```

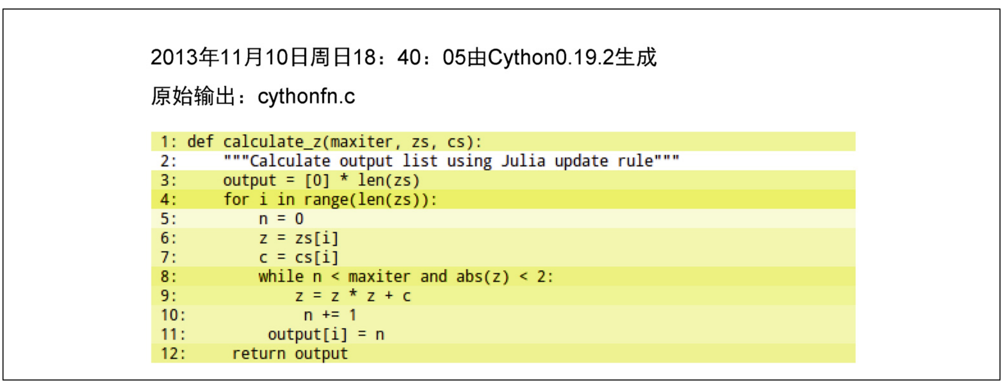
`--inplace` 参数让 Cython 在当前目录中构建编译模块，而不是在一个独立的构建目录中。在构建完成后，我们会有两个难以卒读的中间文件 `cythonfn.c` 和 `calculate.so`。

现在当运行 julia1.py 时，会导入编译模块，在 Ian 的笔记本电脑上计算 Julia 集在 8.9 秒内做完，而不是通常的多于 11 秒。这是一个花费很少工作量的小小改进。

7.6.2　Cython 注解来分析代码块

前面的示例给我们演示了我们能够快速构建一个编译模块。对紧凑的循环和数学运算来说，这往往能带来速度提升。很明显，我们不应该盲目优化——我们需要知道哪里慢了，这样就能够决定集中于哪个方向去努力。

Cython 有一个注解选项能输出一个可以让我们在浏览器上查看的 HTML 文件。要产生注解，我们要使用命令 `cython -a cythonfh.pyx` 来产生输出文件 cythonfn.html。在浏览器中它看起来就像图 7-2 中的那样。一张类似的图片在 Cython 文档中有提供。



2013年11月10日周日18：40：05由Cython0.19.2生成

原始输出：cythonfn.c

```
1: def calculate_z(maxiter, zs, cs):
2:     """Calculate output list using Julia update rule"""
3:     output = [0] * len(zs)
4:     for i in range(len(zs)):
5:         n = 0
6:         z = zs[i]
7:         c = cs[i]
8:         while n < maxiter and abs(z) < 2:
9:             z = z * z + c
10:            n += 1
11:        output[i] = n
12:    return output
```

图 7-2　彩色版 Cython 的非注解函数输出

每一行能够被双击扩展来显示生成的 C 代码。更多的黄色意味着“更多的 Python 虚拟机调用”，而更多的白色意味着“更多的非 Python 的 C 代码”。目标就是移除尽可能多的黄色并以更尽可能多的白色来结束。

尽管“更多的黄色线条”意味着更多的虚拟机调用，但这并不一定让你的代码跑得更慢。每一个虚拟机调用都有一个开销，但是这些调用的开销仅当发生于大循环的内部时才会变得显著。在大循环外部的调用（例如，在函数开始处创建输出的一行）相对于内部的计算循环的开销来说并不高。不用把你的时间浪费在不会拖慢运行速度的代码行间。

在我们的例子中，那些回调 Python 虚拟机最多次的代码行（“最深的黄色”）是第 4 行和第 8 行。从我们之前的剖析工作中，我们知道第 8 行可能被调用了超过 3 千万次，所以这是一个需要集中很多精力的候选对象。

第 9、10 和 11 行几乎是一样深度的黄色，我们也知道它们是在紧凑的内循环内。总体上，它们占了这个函数执行时间的大部分，所以我们需要首先集中于它们上面。如果你需要回忆起多少时间花费在这部分上了，请向前参考 2.8 节。

第 6 和第 7 行的黄色深度减弱了，既然它们只是被调用了一百万次，它们对最终的速度影响就小得多了，所以我们可以在稍后再集中于它们身上。事实上，因为它们是列表对象，我们几乎没有办法来提升他们的存取速度，除非如我们将会在 7.8 节中读到的那样，用 `numpy` 数组来取代 `list` 对象，这会带来小小的速度优势。

为了更好地理解黄色区域，我们可以双击展开每一行。在图 7-3 中，我们可以看到为了创建输出列表，我们遍历了 `zs` 的长度，创建了新的由 Python 虚拟机来引用计数的 Python 对象。尽管这些调用是耗时的，它们不会真正影响这个函数的执行时间。

为了改进函数的执行时间，我们需要开始声明对象的类型，这些对象被包含在了耗时的内循环中。这样这些循环才能够减少相对耗时的回调 Python 虚拟机的次数，从而节省时间。

一般情况下，可能最消耗 CPU 时间的代码行是下面这些：

- 在紧凑的内循环内。
- 解引用 `list`、`array` 或者 `np.array` 这些项。
- 执行数学运算。



2013年11月10日周日18：40：05由Cython 0.19.2生成

原始输出：cythonfn.c

```
1: def calculate_z(maxiter, zs, cs):
2:     """Calculate output list using Julia update rule"""
3:     output = [0] * len(zs)

  /* "cythonfn.pyx":3
 * def calculate_z(maxiter, zs, cs):
 *     """Calculate output list using Julia update rule"""
 *     output = [0] * len(zs)             # <<<<<<<<<<<<<<
 *     for i in range(len(zs)):
 *         n = 0
 */
  __pyx_t_1 = PyObject_Length(__pyx_v_zs); if (unlikely(__pyx_t_1 == -1)) {__pyx_filename = __pyx_f[0];
  __pyx_t_2 = PyList_New(1 * ((__pyx_t_1<0) ? 0:__pyx_t_1)); if (unlikely(!__pyx_t_2)) {__pyx_filename =
  __Pyx_GOTREF(__pyx_t_2);
  { Py_ssize_t __pyx_temp;
    for (__pyx_temp=0; __pyx_temp < __pyx_t_1; __pyx_temp++) {
      __Pyx_INCREF(__pyx_int_0);
      PyList_SET_ITEM(__pyx_t_2, __pyx_temp, __pyx_int_0);
      __Pyx_GIVEREF(__pyx_int_0);
    }
  }
  __pyx_v_output = ((PyObject*)__pyx_t_2);
  __pyx_t_2 = 0;

4:     for i in range(len(zs)):
5:         n = 0
6:         z = zs[i]
7:         c = cs[i]
8:         while n < maxiter and abs(z) < 2:
9:             z = z * z + c
10:            n += 1
11:         output[i] = n
12:     return output
```

图 7-3　在一行 Python 代码后面的 C 代码

备忘

如果你并不知道哪些代码行执行得最频繁，使用一个剖析工具是最合适的——`line_profile`，在 2.8 节中讨论过。你将会得到哪些行在 Python 虚拟机中消耗最多，这样你就会有一个清晰的依据再次集中于那些代码行以获得最佳的速度。

7.6.3　增加一些类型注解

图 7-2 展示了函数几乎每一行都回调了 Python 虚拟机。我们所有的数值运算也都回调了 Python 虚拟机，因为我们使用的是高层的 Python 对象。我们需要把这些都转换为本地 C 对象，接着在进行数值方面的编码时，我们需要把结果再转换回 Python 对象。

在例 7-6 中，我们会看到怎样使用 `cdef` 语法来增加一些原始类型。

备忘

值得重视的是那些类型只能被 Cython 理解，而不是 Python。Cython 使用这些类型把 Python 代码转换为 C 对象，而不需要回调 Python 栈，这意味着运算能执行得更快，但是损失了灵活性和开发速度。

我们增加的类型是：

- 有符号整数 `int`。
- 只能为正的无符号整数 `unsigned int`。
- 双精度复数 `double complex`。

`Cdef` 关键字让我们在函数体内声明变量。作为 C 语言规范的需求，这些声明必须出现在函数顶部。

例 7-6　增加原始 C 类型开始让我们的编译函数运行更快，通过用 C 做更多工作从而减少 Python 虚拟机的工作

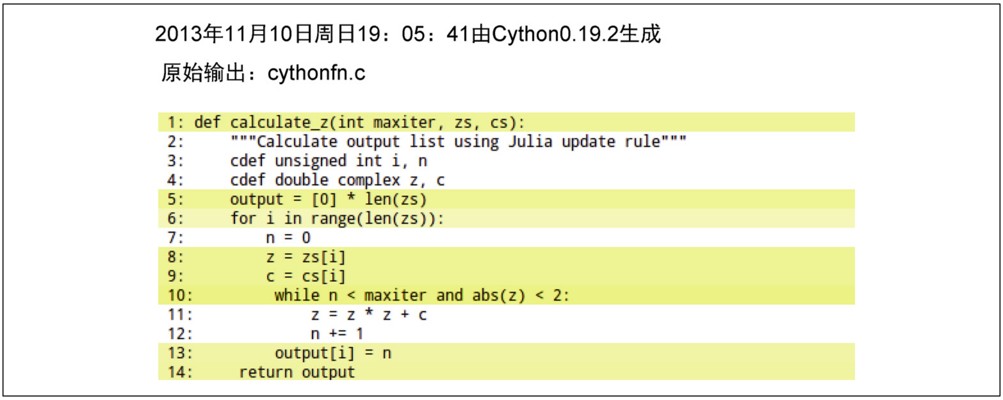
```
def calculate_z(int maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, n
    cdef double complex z, c
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

备忘

当增加 Cython 注解后，你正在给.pyx 增加非 Python 的代码。这意味着你丧失了在解释器中开发 Python 的天然的交互性。对熟悉 C 语言编程的人来说，我们又回到了编码—编译—运行—调试的循环中去了。

你可能会奇怪我们是否可以给作为参数传入的 list 增加类型注解。我们能够使用 `list` 关键字，但是实践中对于这个例子无效。`list` 对象还是不得不在 Python 层面去查询来找出其中的内容，速度是很慢的。

给其中一些原始对象增加类型的做法在图 7-4 中的注解输出中反映了出来。第 11 和 12 行是很关键的地方——这 2 行我们调用最频繁的代码——现在已由黄色转为了白色，显示出它们不再去回调 Python 虚拟机了。我们可以期待与前面的版本比较之下的一个很大的速度提升。第 10 行被调用了超过 3 000 万次，所以我们还是要集中在它上面。



2013年11月10日周日19：05：41由Cython0.19.2生成

原始输出：cythonfn.c

```
def calculate_z(int maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, n
    cdef double complex z, c
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

图 7-4　我们的第一个类型注解

在编译后，这个版本花了 4.3 秒来完成任务。只对函数做了很少的改动，我们却跑出了是原来 Python 版本两倍的速度。

值得重视的是我们获得提速的原因是更多的频繁调用的运算被放到了 C 的层面——在这个案例中，更新了 `z` 和 `n`。这意味着 C 编译器能够去优化更底层的函数来对代表这些变量的字节做运算，而不是去调用相对慢速的 Python 虚拟机。

我们在图 7-4 中可以看到 `while` 循环还是相当耗时的（黄色部分）。耗时的调用在于 Python 对复数 `z` 的 `abs` 函数中。Cython 没有对复数提供原生的 `abs` 函数。作为替代，我们可以提供自己的本地扩展。

就如本章前面提醒的那样，对一个复数做 `abs` 涉及对实部和虚部的平方和开平方根。在我们的测试中，我们想要看看结果的平方根是否小于 2。与其开平方根，我们不如对比较表达式的另一端求平方，因此我们把`<2` 转换为`<4`。这就避免了必须要计算平方根来作为 `abs` 函数的最终结果。

实质上，我们从下式开始：

$$\sqrt{c.real^2 + c.imag^2} < \sqrt{4}$$

我们还有简化版的运算：

$$c.real^2 + c.imag^2 < 4$$

如果我们在下列代码中保留了 `sqrt` 运算，我们还是能看到执行速度的提升。优化代码的秘诀之一就是让它尽可能的少干活。通过考虑一个函数的最终目标来移除相

对耗时的运算意味着 C 编译器能集中于它所擅长的方面，而不是尝试去感知程序员的最终需求。

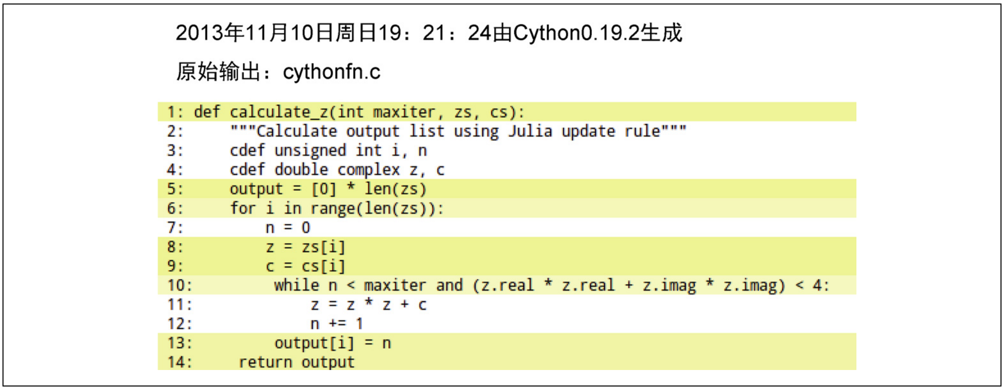
编写等价的但是更特殊的代码来解决相同的问题就是所谓的强度减弱。你用更糟糕的灵活性（和可能更糟糕的可读性）去换来更快的执行速度。

数学分解在下一个例子中，见例 7-7，其中我们已经用一行简化的扩展数学表达式来代替相对耗时的 abs 函数。

例 7-7　用 Cython 来扩展 abs 函数

```
def calculate_z(int maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, n
    cdef double complex z, c
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

通过代码注解，我们看到在第 10 行（图 7-5）以 while 语句为代价换来的小小改进。现在它包含了更少的 Python 虚拟机调用。尽管对于能够获得多少速度提升来说并不是立即显而易见的，但是我们知道这行被调用了超过 3000 万次，因此我们期待一个优越的性能提升。


2013年11月10日周日19：21：24由Cython0.19.2生成

原始输出：cythonfn.c

```
1: def calculate_z(int maxiter, zs, cs):
2:     """Calculate output list using Julia update rule"""
3:     cdef unsigned int i, n
4:     cdef double complex z, c
5:     output = [0] * len(zs)
6:     for i in range(len(zs)):
7:         n = 0
8:         z = zs[i]
9:         c = cs[i]
10:         while n < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
11:             z = z * z + c
12:             n += 1
13:         output[i] = n
14:     return output
```

图 7-5　展开数学表达式取得了最终的胜利

这个改变有巨大的效果——通过减少在最内循环中 Python 调用的次数，我们大大降低了函数的运算时间。这个新版本只用 0.25 秒就执行完毕了，具有超过原版本 40 倍的速度提升，令人惊叹。

备忘

Cython 支持几种编译成 C 的方式，有一些比这里描述的全类型注解方式要来得简单。如果你想要一个更简单的起点来使用 Cython，并且查看 `pyximport` 来方便向同事介绍 Cython，你应该熟悉纯 Python 模式。

为了对代码片段做最终可能的改进，我们会禁止 list 中的每个解引用的边界检查。边界检查的目的是确保程序不会去访问超出分配数组的空间——在 C 中，很容易无意中访问了超出数组边界的内存，产生不可预料的结果（可能是一个段错误!）。

Cython 默认会保护程序员免于意外地去寻址超出 list 边界的空间。这种保护消耗了一点 CPU 时间，但是它发生于我们函数的外循环处，所以总体上不会占用很多时间。通常禁止边界检查是安全的，除非你要执行自己的计算来做数组寻址，这种情况下你会不得不小心翼翼地呆在 list 的边界内。

Cython 有一系列标志开关可以用各种各样的方式去表述。最简单的就是在.pyx 文件的起始处把它们作为单行注释加入。也可以利用装饰器或编译时标志来改变这些设定。为了禁止边界检查，我们在.pyx 文件开头的注释内给 Cython 增加了一行指令（directive）。

```
#cython: boundscheck=False
def calculate_z(int maxiter, zs, cs):
```

要注意的是，禁止边界检查只会节省一点时间，因为它发生于外循环中，而不是在更耗时的内循环中。对于这个例子来说，它不会节省更多的时间。

备忘

如果你的 CPU 密集型代码在频繁解引用的循环中，尝试禁止边界检查和外围检查。

7.7 Shed Skin

Shed Skin 是一个和 Python2.4～2.7 一起使用的实验性的把 Python 转为 C++的编译

器。它使用类型引用来自动检查 Python 程序来注解每一个变量。被注解的代码接着就会被翻译成 C 代码，可以被 g++之类的标准编译器编译。自动反射是 Shed Skin 的一个很有趣的特征，用户只要求提供一个关于怎样用正确类型的数据来调用函数的例子，Shed Skin 会把剩下的都给做了。

类型引用的好处是程序员不需要显式地手动声明类型。付出的开销就是分析器需要能够推导出程序中的每个变量的类型。在当前版本中，上千行 Python 代码能被自动转换成 C。它使用了 Boehm 的标记-清除垃圾收集器来自动管理内存。Boehm 的垃圾收集器也用于 Mono 和 Java 的 GNU 编译器。Shed Skin 的缺点就是它使用了外部实现的标准库。任何没有被实现的库（包含 numpy）将不受支持。

整个项目有超过 75 个例子，包括了许多集中于数学方面的纯 Python 模块，甚至有一个能够完整工作的 Commodore 64 模拟器。每一个模块在用 Shed Skin 编译后，相比本地运行的 CPython，速度得到明显的提升。

Shed Skin 能构建独立的可执行程序，不依赖于已安装的 Python 安装包或者在常规 Python 代码中用 import 导入的扩展模块。

编译模块管理它们自己的内存。这意味着来自 Python 进程的内存被拷贝进来并且结果被拷贝出去——没有显式的共享内存。对于大块内存（例如，一个大矩阵）执行拷贝的开销很显著，我们在本节末尾会简略看一下。

Shed Skin 提供了和 PyPy 一样的很多好处（请看 7.11 节）。因此，PyPy 可能更容易使用，因为它不需要任何编译步骤。Shed Skin 自动增加类型注解的方式可能让一些用户很感兴趣，并且生成的 C 代码比 Cython 生成的可读性高，如果你希望修改生成的 C 代码的话。我们肯定地猜测自动类型反射代码会让社区中的其他编译器编写者很感兴趣。

7.7.1 构建扩展模块

在这个例子中，我们会构建一个扩展模块。我们会导入生成的模块，就如在 Cython 的示例中做的那样。我们也能够把这个模块编译成一个独立的可执行文件。

在例 7-8 中，我们有一个在独立模块中的代码示例，它包含了无法注解的普通 Python 代码。也要注意我们已经增加了一个__main__测试——这使得这个模块可以被独立运行来做类型分析。Shed Skin 能够使用这个__main__块来提供示例中的参数，从而去推导出传入 calculate_z 的参数类型，进而推导出在 CPU 密集型函数内部所使用的数据类型。

例 7-8　把我们的 CPU 密集型函数挪到一个分离的模块去（就如我们用 Cython 做的那样）来让 Shed Skin 的自动类型推导系统运行

```
# shedskinfn.py
def calculate_z(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output


if __name__ == "__main__":
    # make a trivial example using the correct types to enable type inference
    # call the function so Shed Skin can analyze the types
    output = calculate_z(1, [0j], [0j])
```

我们能够像例 7-9 中那样，既可以在这个模块被编译之前导入它，也可以在这个模块被编译之后再导入它，就像通常所做的那样。既然代码没有被改动过（不像使用 Cython 那样），我们就能在编译前调用 Python 的原生模块。如果你还未编译过你的代码，你不会得到速度提升，但是你能够使用常规的 Python 工具以一种轻量级的方式来调试。

例 7-9　导入外部模块以便让 Shed Skin 编译它

```
...
import shedskinfn
...
def calc_pure_python(desired_width, max_iterations):
    #...
    start_time = time.time()
    output = shedskinfn.calculate_z(max_iterations, zs, cs)
    end_time = time.time()
    secs = end_time - start_time
    print "Took", secs, "seconds"
...
```

就如在例 7-10 中所见，我们能使用 `Shedskin -ann shedskin.py` 让 Shed Skin 提供一个它所分析的注解输出，这会生成 `shedskinfn.ss.py`。如果我们要编译一个扩展模块，我们只需要用空的`__main__`函数来做分析“种子”。

例 7-10　检查 Shed Skin 的注解输出看下它推导出了哪些类型

```
# shedskinfn.ss.py
def calculate_z(maxiter, zs, cs):          # maxiter: [int],
                                           # zs: [list(complex)],
                                           # cs: [list(complex)]
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)                 # [list(int)]
    for i in range(len(zs)):               # [__iter(int)]
        n = 0                              # [int]
        z = zs[i]                          # [complex]
        c = cs[i]                          # [complex]
        while n < maxiter and abs(z) < 2: # [complex]
            z = z * z + c                  # [complex]
            n += 1                         # [int]
        output[i] = n                      # [int]
    return output                          # [list(int)]


if __name__ == "__main__":                 # []
    # make a trivial example using the correct types to enable type inference
    # call the function so Shed Skin can analyze the types
    output = calculate_z(1, [0j], [0j])    # [list(int)]
```

__main__的类型被分析之后，接着在 calculate_z 的内部，变量 z 和 c 的类型能够从与它们有互动的对象中推导出来。

我们使用 shedskin --extmod shedskinfn.py 来编译这个模块，会生成下列文件：

- shedskinfn.hpp（C++头文件）。
- shedskinfn.cpp（C++源文件）。
- Makefile。

通过运行 make，我们就生成了 shedskinfn.so。我们能够通过 import shedskinfn 在 Python 中使用。用 shedskinfn.so 来执行 julia1.py 的时间只有 0.4 秒——相比未编译的版本，只做了很小的工作就取得了巨大的胜利。

我们也能展开 abs 函数，就如我们用 Cython 在例 7-7 中做的那样。在运行这个版本（只改了 abs 那一行）并使用一些额外的标志位--nobounds --nowrap 之后，我们得到了一个 0.3 秒的最终执行时间。这比 Cython 版本稍慢一点（慢 0.05 秒），但是我们不需要声明所有的类型信息。这使得用 Shed Skin 来做实验很容易。PyPy 以相近的速度运行了这个代码的相同版本。

备忘

仅仅是因为 Cython、PyPy 和 Shed Skin 在这个例子中跑出了相近的速度，但这并不意味着这是普遍性的结果。为了让你的项目得到最佳提速，你必须调查这些不同的工具并运行你自己的实验。

Shed Skin 允许你声明额外的编译期选项，比如`-ffast-math`或`-o3`，并且你能够在两遍扫描（第一遍收集执行统计信息，第二遍在这些统计信息基础上优化生成的代码）中增加剖析导向优化（PGO）来设法做出进一步的速度提升。然而剖析导向优化没有让 Julia 的例子跑得更快，在实践中它通常很少或没有真实效果。

你应该注意到默认整数是 32 比特的，如果你想要更大的 64 比特整数范围，那么就声明`--long`标志。你也应该避免在紧凑的内循环中分配小对象（例如，new tuples），因为垃圾收集器并不能高效地处理它们。

7.7.2 内存拷贝的开销

在我们的示例中，Shed Skin 通过把数据扁平化成基本 C 类型的方式把 Python 的 list 对象拷贝进 Shed Skin 的环境中来，接着在执行函数的末尾把结果从 C 函数转换为 Python 的`list`。这些转换和拷贝花费时间。这大概就占了我们在前面的结果中看到的多出来的 0.05 秒（相比 Cython 的结果多了 0.05 秒）吧？

我们能够修改 Shedskinfn.py 文件来移除实际的工作量，这样我们就能计算经由 Shed Skin 把数据拷进拷出的开销。下面的`calculate_z`的变体是我们所需要的：

```
def calculate_z(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)
    return output
```

当我们用这个框架函数执行`julia1.py`时，执行时间接近于 0.05 秒（显然它不计算正确结果！）。这个时间是把 2000000 个复数拷贝进`calculate_z`并把 1000000 个整数再次拷贝出来的开销。实质上，Shed Skin 和 Cython 生成了相同的机器码，执行速度的差异归结于 Shed Skin 在一个独立的内存空间运行，还有就是所需的拷进拷出数据的开销。硬币的另一面就是，使用 Shed Skin，你不必预先做注解工作，节省了相当多的时间。

7.8 Cython 和 numpy

list 对象（作为背景，请看第 3 章）的每一个解引用都有开销，因为它们所引用的对象可以存在于内存中任意处。而相反，array 对象在 RAM 的连续块中存储原生类

型，能够被快速寻址。

Python 具有 array 模块，为基础原生类型（包括整数、浮点数、字符串和 Unicode 字串）提供了一维存储。Numpy 的 numpy.array 模块允许多维存储和更多样的原生类型，包括复数。

当以可预测的方式去迭代访问一个 array 对象时，如果要移动到序列中的下一个原生项，编译器会被指导去直接访问该项的内存地址，而不是让 Python 来计算出一个合适的地址。既然数据布局到了连续块中，在 C 中用偏移量来计算下一项的地址就是轻而易举的，而不要让 CPython 去计算来得到相同的结果，因为这会涉及慢速的虚拟机回调。

你应该注意到了如果运行接下来的 numpy 版本而不用任何 Cython 注解（例如，只是作为一个普通 Python 脚本运行），大概要 71 秒跑完——远超出普通的 Python list 脚本，它大概花费 11 秒。拖慢运行速度的原因是在 numpy lists 中解引用每一个元素的开销——它从来不是为这种用法而设计的，即使对初学者来说，这种用法看上去直观。通过编译代码，我们移除了这个开销。

Cython 为此有两种特殊的语法形式。更老的 Cython 版本有一种特殊的访问 numpy array 的类型，但最近更一般化的缓存接口协议通过 memoryview 引入了进来——允许对实现了缓存接口的任意对象进行相同的低级访问，包括 numpy arrays 和 Python arrays。

缓存接口的一个附加优势是内存块能够很容易地在其他 C 库中共享，而不需要把它们从 Python 对象转换成其他形式。

例 7-11 中的代码块看上去有一点像原来的实现，除了我们增加的 memoryview 注解外。函数的第 2 个参数是 double complex[:] zs，意味着我们有一个使用缓存协议（用[]声明）的双精度复数对象，包含了一个一维的数据块（由一个冒号：声明）。

例 7-11　Julia 计算函数的注解 numpy 版本

```
# cython_np.pyx
import numpy as np
cimport numpy as np

def calculate_z(int maxiter, double complex[:] zs, double complex[:] cs):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, n
    cdef double complex z, c
    cdef int[:] output = np.empty(len(zs), dtype=np.int32)
```

```
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

除了使用缓存注解语法声明输入参数外，我们也注解了输出变量，由 empty 给它分配了一个一维的 numpy array。调用 empty 会分配一块内存，但是不会用完整的数值初始化内存，所以它不包含任何东西。我们在内循环中会覆写这个 array 的内容，所以我们不需要重复给它赋默认值。这要比分配并给 array 的内容赋默认值要稍快一点。

我们也使用更快、更显式的数学版本来展开 abs 的调用。这个版本用了 0.23 秒跑完——结果比原来在例 7-7 中的纯 Python 的 Julia 例子的 Cython 化版本要稍微快一点。纯 Python 版本有每次解引用一个 Python 复数对象的开销，但是这些解引用发生于外循环中，所以不占用多少执行时间。在外循环之后，我们做了这些变量的本地版本，它们以“C 的速度”运行。这个 numpy 的例子和之前的纯 Python 例子中的内循环都对相同的数据做了相同的处理，所以执行时间的差异归因于外循环中的解引用和输出队列的创建。

在一台机器上使用 OpenMP 来做并行解决方案

作为这版代码演进的最后步骤，让我们看一下使用 OpenMP C++扩展来并行化处理让我们为难的并行问题。如果你的问题适合这个模式，那么你就能很快发挥你的计算机多核的优势。

OpenMP（Open Multi-Processing）是一个定义良好的跨平台 API，支持并行执行，以及与 C、C++和 Fortran 的内存共享。它被构建入了大多数的现代 C 编译器，如果 C 代码编写合适的话，并行化就会在编译器级别上发生，所以它就给开发者使用 Cython 带来了相对小的工作量。

与 Cython 一起，OpenMP 能够通过使用 prange（并行 range）操作符和给 setup.py 增加-fopenmp 编译指令的方式加入进来。在一个 prange 循环中的工作就能够做到并行运行，因为我们禁止了全局解释器锁（GIL）。

一个修改过的支持 prange 的代码版本显示在例 7-12 中。用 nogil:来声明禁止 GIL 的代码块，在这个代码块内部，我们使用 prange 为循环开启 OpenMP 并行模式来独立计算每一个 i。

警告

当禁止 GIL 时，我们一定不能在常规 Python 对象（例如，lists）上操作，必须要在原生对象和支持 memoryview 接口的对象上去操作。如果并行操作了常规的 Python 对象，我们不得不去解决随之而来的内存管理问题，而这是 GIL 意图避免的。Cython 不阻止我们去操控 Python 对象，但是如果你这样做，只会招来痛苦和困扰。

例 7-12　增加 prange 来启用 OpenMP 并行化

```
# cython_np.pyx
from cython.parallel import prange
import numpy as np
cimport numpy as np

def calculate_z(int maxiter, double complex[:] zs, double complex[:] cs):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, length
    cdef double complex z, c
    cdef int[:] output = np.empty(len(zs), dtype=np.int32)
    length = len(zs)
    with nogil:
        for i in prange(length, schedule="guided"):
            z = zs[i]
            c = cs[i]
            output[i] = 0
            while output[i] < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
                z = z * z + c
                output[i] += 1
    return output
```

为了编译 cython_np.pyx，我们不得不修改 setup.py 脚本，就如在例 7-13 中显示的那样。我们让它通知 C 编译器在编译期间使用`-fopenmp` 作为参数来启用 OpenMP 以及和 OpenMP 库去链接。

例 7-13　为 Cython 在 setup.py 中增加 OpenMP 编译器和和链接器标志

```
#setup.py
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
        cmdclass = {'build_ext': build_ext},
        ext_modules = [Extension("calculate",
                                 ["cython_np.pyx"],
```

```
                                    extra_compile_args=['-fopenmp'],
                                    extra_link_args=['-fopenmp'])]
        )
```

使用 Cython 的 prange，我们能够选择不同的调度方式。使用 static，工作负载可以均匀地在可用的 CPU 之间分布。我们的一些计算区域在时间上是开销很大的，另一些则开销很小。如果我们用 static 让 Cython 在 CPU 之间平等地调度工作块，那么一些区域要比另外一些完成得快，那些线程就会处于空闲状态。

Dynamic 和 guided 调度选项都企图缓解这个问题，可以通过在运行时动态地把工作分配给更小的块，这样当工作负载的运算时间可变时，CPU 会更均匀地得到分布。对你的代码来说，正确的选择将会是根据你的工作负载的本质而做改变。

通过引入 OpenMP 和使用 schedule=″guided′，我们把执行时间降低到了接近 0.07 秒——guided 调度会动态地分配工作，所以更少的线程在等待新的工作。

我们也可能为这个例子使用#cython:boundscheck=False 来禁止边界检查，但是这不会改进我们的运行时间。

7.9 Numba

来自 Continuum Analytics 的 Numba 是一个专用于 numpy 代码的即时编译器，在运行时由 LLVM 编译器（不是由我们在之前例子中所用的 g++或 gcc）来编译。它不需要预编译扫描，所以当你用它来运行新代码时，它会根据你的硬件编译每一个注解的函数。漂亮之处就是你提供了一个装饰器告诉它需要集中于哪些函数，接着你就让 Numba 接手。它的目标是在所有标准 numpy 代码上运行。

这是一个更年轻的项目（我们使用 v0.13），它的 API 可能会随着每一次发布发生小小的改变，所以认为当前它在研究环境中更有用。如果你使用 numpy array，并且有迭代许多次的非向量代码，Numba 会给你一个快速而不需付出什么代价的胜利。

使用 Numba 的一个缺点就是工具链——它使用了 LLVM，具有很多的依赖性。我们推荐你使用 Continuum 的 Anaconda 发布包，因为它提供了所有一切，否则在一个全新的环境中安装 Numba 是一个很耗时的任务。

例 7-14 展示了给我们的核心 Julia 函数增加@jit 装饰器。这就是所需的全部了，numba 被导入的事实意味着 LLVM 机制会在运行时发挥作用，在幕后编译这个函数。

例 7-14　让函数应用@jit 装修器

```
from numba import jit
...
@jit()
def calculate_z_serial_purepython(maxiter, zs, cs, output):
```

如果移除了`@jit`装饰器，那么这只是用 Python2.7 运行的 Julia 演示版的`numpy`版本，它花费了 71 秒。增加`@jit`装饰器后执行时间降低到 0.3 秒。这已经很接近我们用 Cython 达成的结果了，然而又没有多出注解的工作量。

如果我们在同一个 Python 会话中第 2 次去运行相同的函数，它甚至跑得更快——如果参数类型一样的话就不需要去编译目标函数了，所以整体执行速度就更快。第 2 次 Numba 的运行结果等价于我们之前一起使用 Cython 和`numpy`获得的结果(所以它几乎不需要做什么就可以达到与 Cython 一样快的结果!)。PyPy 有同样的热身需要。

当用 Numba 去调试时，值得注意的是你可以让 Numba 去显示在一个编译函数内部的正在处理的变量类型。在例 7-15 中，我们可以看到`zs`被 JIT 编译器识别为一个复杂的数组。

例 7-15　调试推导出的类型

```
print("zs has type:", numba.typeof(zs))
    array(complex128, 1d, C))
```

Numba 也支持其他形式的自省，比如`inspect_types`，从而让你可以检查编译代码去看看类型信息在哪里被推导出来。如果类型丢失了，那你就能细化函数的表达方式去帮助 Numba 得到发现更多的类型推导的机会。

Numba 的高级版本，NumbaPro，可以实验性的用 OpenMP 支持`prange`并行操作符。实验性的 GPU 之处也有。这个项目的目标是很轻松地把使用 numpy 的更慢的 Python 循环代码转换为运行快速的代码，能在 CPU 或 GPU 上运行，这个可以关注一下。

7.10　Pythran

Pythran 是一个把 Python 转换成 C++的编译器，作用于包含对部分`numpy`支持的 Python 子集。它表现得有点像 Numba 和 Cython——你注解一个函数的参数，接着它接管进一步的类型注解和代码特化。它利用了矢量化可能性和基于 OpenMP 的并行化可能性。它只用 Python2.7 运行。

Pythran 的一个很有趣的特点是它会意图自动发现并行化的机会（例如，如果你正使用一个 map），并把它转换成并行代码，而不需要你额外的工作量。你也能用 pragma omp 来声明并行区域，在这方面，感觉它很类似于 Cython 对 OpenMP 的支持。

在幕后，Pythran 会企图把通常的 Python 代码和 numpy 代码很激进地编译成速度很快的 C++代码——甚至比 Cython 的结果更快。你应该注意到这个项目是年轻的，你可能会遇到错误；你也应该注意到开发团队很友好并倾向于在几小时内修正错误。

再看一下例 6-9 中的扩散方程。我们已经把例程中的计算部分抽取出来放到一个独立模块中，这样它就能被编译成一个二进制库。Pythran 的一个良好特点就是我们不产生与 Python 不兼容的代码。回想起 Cython，我们不得不用注解的 Python 来创建 .pyx 文件，这样就不能被 Python 直接运行了。使用 Pythran 时，我们只需添加一行注释来让 Pythran 编译器识别。这意味着如果我们删除了生成的.so 编译模块，我们就能单独用 Python 来运行我们的代码——这对调试很有利。

在例 7-16 中，你可以看见我们更早期的传热方程的例子。evolve 函数有一行注释来为函数注解类型信息（因为它是注释，所以如果你不用 Pythran 运行，Python 只会忽略注释）。当 Pythran 运行时，它看到那行注释并且通过每一个相关的函数来传播类型信息（很像我们见到的 Shed Skin）。

例 7-16　添加一行注释去注解 evolve()的入口点

```
import numpy as np
def laplacian(grid):
    return np.roll(grid, +1, 0) +
                np.roll(grid, -1, 0) +
                np.roll(grid, +1, 1) +
                np.roll(grid, -1, 1) - 4 * grid

#pythran export evolve(float64[][], float)
def evolve(grid, dt, D=1):
    return grid + dt * D * laplacian(grid)
```

我们可以用 pythran diffusion_numpy.py 来编译这个模块，会输出 diffusion_numpy.so。从一个测试函数中，我们可以导入这个新模块并且调用 evolve。在 Ian 的笔记本上，不用 Pythran 的话，这个函数在一个 8192 × 2192 的网格上运行了 3.8 秒。用了 Pythran 后降低到 1.5 秒。显然，如果 Pythran 支持你需要的函数的话，它就能够得到令人印象深刻的性能提升，而几乎无须做什么工作。

速度提升的原因是 Pythran 有它自己的 roll 函数版本，功能更少——因此它能编

译成复杂度更低的运行可能更快的代码。这也意味着它比 numpy 版本（Pythran 的作者说明 Pythran 只实现了部分 numpy）的灵活度要低，但是当它工作时，能够胜出我们所见到的其他工具的运行结果。

现在让我们把相同的技术应用于 Julia 的展开数学表达式的例子中。只是给 calculate_z 添加一行注解，我们就能把运行时间降低到 0.29 秒——比 Cython 的结果慢一点点。在外循环的前面添加一行 OpenMP 声明后把执行时间降低到 0.1 秒，距离 Cython 的最佳 OpenMP 结果已经不远。注解的代码在例 7-17 中可见。

例 7-17　得到 OpenMP 支持的注解 Pythran 的 calculate_z

```
#pythran export calculate_z(int, complex[], complex[], int[])
def calculate_z(maxiter, zs, cs, output):
    #omp parallel for schedule(guided)
    for i in range(len(zs)):
```

迄今为止我们所见的技术都涉及使用常规 CPyhon 解释器之外的编译器。现在让我们看一下 PyPy，它提供了一个全新的解释器。

7.11　PyPy

PyPy 是一个 Python 语言的替代实现，包含了一个可跟踪的即时编译器，它兼容 Python2.7，也有实验性的 Python3.2 版本。

PyPy 是一个普适性的 CPython 替代品，提供了几乎所有的内置模块。该项目包含了 RPython 翻译工具链，被用来构建 PyPy（也可用来构建其他解释器）。PyPy 中的 JIT 编译器很高效，能够看到良好的速度提升而几乎或完全不需要你这边的任何工作。看看 12.5 节，这是一个大规模 PyPy 部署成功的故事。

PyPy 运行我们的纯 Python 版的 Julia 演示而不做任何改动。使用 CPython，它用了 11 秒，而使用 PyPy，它用了 0.3 秒。这意味着 PyPy 取得了很接近于例 7-7 中的 Cython 的成果，而完全没有任何工作量——令人印象很深刻！就如我们在讨论 Numba 时观察到的那样，如果在同一会话中再次运行计算，那么第 2 次（以及接下来每次）会比第 1 次刚编译完那会儿要跑得快。

PyPy 支持所有内置模块的事实是很有趣的——这意味着可以像在 CPython 中一样用多进程方式工作。如果你在运行内置模块时遇到问题，你能用多进程方式并行地去运行，除此之外，还会得到你所希望的速度提升。

PyPy 的速度随着时间而演进。图 7-6 中的图表来自 speed.pypy.ord，会告诉你它的成熟度。这些速度测试反映了多种多样的用户案例，不只是数学运算。很清楚 PyPy

提供了比 CPython 更快的体验。

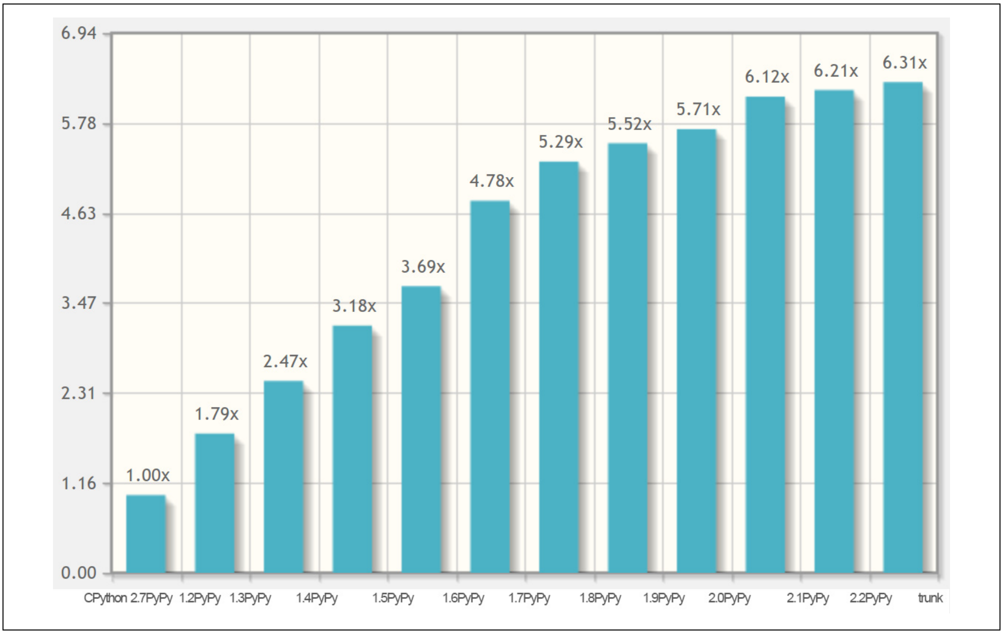


图 7-6　每一个新版本 PyPy 提供了速度提升

7.11.1　垃圾收集的差异

PyPy 使用了不同于 CPython 的垃圾收集器，这能导致你的代码行为产生不明显的改变。尽管 CPython 使用引用计数，PyPy 却使用了修改后的标记和清理方法来推迟清除不用的对象。这两者都是对 Python 规范的具体实现，你只需注意当交换时，需要做一些代码改动。

一些在 CPython 中见到的编码方式依赖于引用计数的行为——尤其是冲刷文件（如果这些文件打开后被写入了，而不去显式地关闭文件）。使用 PyPy，相同的代码可以运行，但是对文件的更新可能会推迟冲刷到磁盘上，直到垃圾收集器下次运行时才会被冲刷。一种既能使用 PyPy，又能使用 Python 工作的替代方式就是用 with 来使用上下文管理器去打开和自动关闭文件。在 PyPy 的网站上的“PyPy 与 CPython 的差异”那页列出了细节，垃圾收集器的实现细节在网站上也有。

7.11.2　运行 PyPy 并安装模块

如果你从没有运行过一种替代的 Python 解释器，你可能会受益于一个短小的例子。假设你已经下载并解压出 PyPy，你现在会得到一个包含 bin 目录的文件夹结构。按例 7-18 所示运行它来启动 PyPy。

例 7-18　运行 PyPy 来看看它实现了 Python 2.7.3

```
$ ./bin/pypy
Python 2.7.3 (84efb3ba05f1, Feb 18 2014, 23:00:21)
[PyPy 2.3.0-alpha0 with GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
And now for something completely different: ``<arigato> (not thread-safe, but
well, nothing is)''
```

注意到 PyPy 作为 Python 2.7.3 运行。现在我们需要设置 pip，并想要安装 ipython（注意 IPython 以我们曾见过的相同的 Python 2.7.3 的编译包启动）。如果你不求助于现成的发布包或包管理器来安装 pip，这些显示在例 7-19 中的步骤和你可能已经用 CPython 做过的步骤一模一样。

例 7-19　为 PyPy 安装 pip 用以安装第三方模块，比如 IPython

```
$ mkdir sources # make a local download directory
$ cd sources
# fetch distribute and pip
$ curl -O http://python-distribute.org/distribute_setup.py
$ curl -O https://raw.github.com/pypa/pip/master/contrib/get-pip.py
# now run the setup files for these using pypy
$ ../bin/pypy ./distribute_setup.py
...
$ ../bin/pypy get-pip.py
...
$ ../bin/pip install ipython
...
$ ../bin/ipython
Python 2.7.3 (84efb3ba05f1, Feb 18 2014, 23:00:21)
Type "copyright", "credits" or "license" for more information.

IPython 2.0.0—An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
Help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
```

注意 PyPy 对像 numpy 之类的项目的支持非同寻常（有一个经由 cpyext 的桥接层，但是它太慢了，以致于对 numpy 没什么用），所以不要期待 PyPy 对 numpy 做强力支持。PyPy 的确有一个实验性的 numpy 移植版本，称为“numpypy”（安装指导在 Ian 的博客上），但是当前它没有提供任何有用的速度优势[①]。

如果你需要其他包，只要是纯 Python 的，就可能安装上，而任何依赖于 C 扩展库的就可能无法有效工作。PyPy 没有引用计数的垃圾收集器，任何为 CPython 编译

① 本书写作期间它没有提供任何有用的速度优势。

的包会使用支持 CPython 垃圾收集器的库调用。PyPy 有一个变通方法，但是会增加很多开销。在实践中，去尝试强迫让更老的扩展库直接和 PyPy 一起工作是无效的。PyPy 的建议就是如果有可能就尝试移除任何 C 扩展代码（它或许只是用来让 Python 代码运行更快，现在那是 PyPy 的工作了）。PyPy 的维基维护了一串可兼容的模块列表。

PyPy 的另一个缺点是它使用了很多内存。每个发布版在这方面都有改良，但在实践中，它可能还是使用了比 CPython 更多的内存。然而内存相对便宜，所以用来为性能提升做交换还是有意义的。一些用户也已报道了在使用 PyPy 时有更少的内存占用。如果这仍然对你很重要，那就用具有代表性的数据去做实验。

PyPy 受限于全局解释器锁，但是开发团队正在搞一个叫作软件事务内存（STM）的项目来企图移除 GIL 的必要性。STM 有一点像数据库事务。它是一个并发控制机制，应用于内存访问。如果在相同的内存空间上发生冲突，它能够回滚改动。集成 STM 的目标是能够让高并发系统有一种并发控制的方式，在运算上丧失了一些效率，但是通过不强迫用户处理并发存取控制的所有方方面面来提高程序员的生产率。

推荐的剖析器工具是 `jitviewer` 和 `logparser`。

7.12 什么时候使用每种工具

如果你正工作于数值处理的项目，那么这些技术中的每一种可能都对你有用。表 7-1 总结了主要选项。

如果你的问题适用于可支持函数的严格范围内，Pythran 大概在 numpy 问题上提供了最佳的性能提升，却花费最少的工作量。它也提供了一些简单的 OpenMP 并行项，这也是一个相对年轻的项目。

表 7-1　可选的编译器总结

	Cython	Shed Skin	Numba	Pythran	PyPy
成熟度	Y	Y	_	_	Y
广泛使用性	Y	_	_	_	_
支持 Numpy	Y	_	Y	Y	_
没有间断的代码改动	_	Y	Y	Y	Y
需要有 C 的知识	Y	_	_	_	_
支持 OpenMP	Y	_	Y	Y	Y

Numba 可能提供了快速的性能提升，而几乎不需要什么工作量，但是它也会有在你的代码上可能不会工作得很好的限制。这也是一个相对年轻的项目。

Cython 可能为最广泛的问题集提供了最好的结果，但是它的确需要更多的工作量和交额外的“支持税”，这要归因于它混合使用了 Python 和 C 注解。

如果你不使用 `numpy` 或其他难以移植的 C 扩展，PyPy 是一个强力的选择。

如果你想要编译成 C 并且不使用 `numpy` 或其他外部库，Shed Skin 可能是有用的。

如果你正在部署一个生产工具，那么你可能想要坚持使用理解良好的工具——Cython 应该是你的主要选择，你可能想要查阅在 12.2 节。PyPy 也被用于产品设置（看看 12.5 节）。

如果你工作于少量的数值处理需求，注意 Cython 的缓存接口接受 `arrary.array` 矩阵——这是一个把一块数据传给 Cython 的简单方法，可用来做快速的数值处理而不用添加 `numpy` 作为项目依赖。

整体而言，Pythran 和 Numba 是年轻但是很有前景的项目，而 Cython 十分成熟。PyPy 到现在被视为相对成熟，并且应该一定要有长期的过程来评估。

在 2014 年的一堂 Ian 的课上，一个有能力的学生实现了 Julia 算法的 C 版本，并且失望地看到它执行得比他的 Cython 版本要慢。他透露在 64 位机器上使用了 32 位浮点数——这要比在 64 位机器上使用 64 位双精度数运行得慢。尽管该学生是一个好的 C 程序员，但他不知道这会牵涉到执行速度上的代价。他改变了代码，他的 C 版本代码尽管明显比自动生成的 Cython 版本要短小得多，却跑出了大概一样的速度。编写原始的 C 版本，对比速度，并且想出修正方案，这一系列行为要比一上来就用 Cython 花费更长的时间。

这只是个趣闻轶事，我们没有建议说 Cython 会生成最好的代码，并且胜任的 C 程序员也能够想出怎样让他们的代码比 Cython 生成的版本运行得更快。然而，值得注意的是，假想手写 C 代码会比由 Python 转换成的代码更快也不靠谱。你一定要始终做基准测试并且使用证据来做决定。C 编译器很擅长把代码转换为相对高效的机器码，而 Python 很擅长让你用更易于理解的语言来表达你的问题——明智地结合这两种力量。

7.12.1 其他即将出现的项目

PyData 编译器页列出了一系列高性能编译工具。Theano 是一个更高级的语言，允许使用高维数组上的数学操作符表达式。它与 `numpy` 紧密结合，并且能为 CPU

和 GPU 导出编译过的代码。有趣的是，它已经受到了深度学习 AI 社区的青睐。Parakeet 集中于涉及密集 numpy 数组的编译运算，使用 Python 的子集。它也支持 GPU。

PyViennaCL 是一个 Python 绑定到 ViennaCL 的数值计算和线性代数库。它使用 numpy，支持 GPU 和 CPU。ViennaCL 用 C++编写，为 CUDA、OpenCL 和 OpenMP 生成代码。它支持密集和稀疏的线性代数运算、BLAS 和求解器。

Nuitka 是一个以替代通常的 CPython 解释器为目标的 Python 编译器，具有创建编译过的可执行程序的选项。它支持 Python2.7 的全部，尽管在我们的测试中，它没有为我们普通的 Python 数值测试产生任何明显的性能提升。

Pyston 是该领域最新的加入者。它使用了 LLVM 编译器，被 Dropbox 支持。由于缺少对扩展模块的支持，它可能遭受与 PyPy 所面临的同样的问题，但是有项目计划去解决这个问题。如果这个问题得不到解决，对 numpy 的支持就不现实。

在我们的社区里，我们很幸运拥有各种各样可选的编译器。虽然它们都各有取舍，但它们也提供了许多能力，这样复杂的项目就能充分利用 CPU 和多核架构的力量。

7.12.2 一个图像处理单元（GPU）的注意点

GPU 在当前是个性感的技术，我们选择了把涉及它们的话题拖后直到至少下一版。这是因为该领域在飞速变化，很可能我们现在所说的所有东西到我们读到它时已经改变了。关键的是，它不只是让你改变所写的代码行，而是随着架构的演进，你可能要彻底改变解决问题的方式。

Ian 工作于一个物理学问题，使用 Python 和 PyCUDA，用了一年的 NVIDIA GTX 480 GPU。到年底为止驾驭了 GPU 的全部能力，并且系统比在双核机器上运行相同的函数快了整整 25 倍。双核的变体用 C 写成，使用了一个并行库，为了数据处理，GPU 的变体大部分由 PyCUDA 所包装的 CUDA 的 C 所表达。很快在那之后，GTX 5xx 系列的 GPU 问世了，许多应用于 4xx 系列的优化就发生了改变。大概耗费一年的工作最终被放弃了，还是倾向于运行在 CPU 上的更易于维护的 C 的解决方案。

这是一个孤立的例子，但是它强调了写底层 CUDA（或 OpenCL）代码的危险。在 GPU 之上的提供了更高级功能的库可能已得到了广泛使用（例如，提供了图像分析或视频转码接口的库），我们规劝你去考虑这些库，而不是去直接为 GPU 编码。

以为你管理 GPU 为目标的项目包括 Numba、Parakeet 和 Theano。

7.12.3 一个对未来编译器项目的展望

在当前的可选的编译器之中，我们已经有一些强大的技术组件。就个人而言，我们想要看到 Shed Skin 的注解引擎变得广泛适用，这样它就能和其他工具一起工作——例如，当学用 Cython（尤其当使用 numpy 时）时，产生一个兼容 Cython 的输出来让学习曲线变平滑。Cython 是成熟的并且紧密集成入 Python 和 numpy，如果学习曲线和对支持的需求不那么可畏，那么更多人会使用它。

更长期的愿望就是看到一些类似 Numba 和 PyPy 的解决方案在常规 Python 代码和 numpy 代码上都提供 JIT 的解决方案。当前没有一个解决方案可用，一个解决了这个问题的工具将会是替代我们当前都在使用的常规 CPython 解释器的强有力的竞争者，而不需要开发者修改他们的代码。

友善的竞争和对新思路的巨大市场真的会让我们的生态系统变成一个富足的地方。

7.13 外部函数接口

有时候自动化解决方案不起作用，你需要自己写一些定制的 C 或 Fortran 代码。这是有可能的，因为编译手段没有找到一些潜在的优化，或者因为你想要利用在 Python 中没有的库或语言特色。在所有这些情况中，你会需要使用外部函数接口，让你去访问用其他语言编写和编译的代码。

在本章其余部分，我们试图用一个外部库来解决 2 阶扩散方程，就以我们在第 6 章[①]中所做的相同的方式。显示在例 7-20 中的这个库代码，能够代表你已经安装的库，或者一些你已经编写的代码。我们要看的方法发挥了巨大的作用，来提取出你的部分代码并把它们挪到另一种语言中去，以便做很目标化的基于语言的优化。

例 7-20　解决 2 阶扩散问题的 C 代码示例

```
void evolve(double in[][512], double out[][512], double D, double dt) {
     int i, j;
    double laplacian;
    for (i=1; i<511; i++) {
       for (j=1; j<511; j++) {
          laplacian = in[i+1][j] + in[i-1][j] + in[i][j+1] + in[i][j-1]\
             - 4 * in[i][j];
          out[i][j] = in[i][j] + D * dt * laplacian;
       }
    }
}
```

① 为了简化，我们不会实现边界条件。

为了使用这个代码，我们必须把它编译成一个创建了.so 文件的共享模块。我们能使用 gcc（或任何其他 C 编译器）通过以下步骤来做：

```
$ gcc -O3 -std=gnu99 -c diffusion.c
$ gcc -shared -o diffusion.so diffusion.o
```

我们能把这个最终的共享库文件放置于可以被我们的 Python 代码所访问的任何地方，但是标准*nix 组织把共享库存放于/usr/lib 和/usr/local/lib。

7.13.1 ctypes

cPython[①]中最基本的外部函数接口是通过 ctypes 模块。这个模块的特性就是具有很多限制性——你要负责做一切，并且需要用一段时间来确认你是按顺序做了这一切。这个额外级别的复杂性在我们的 ctypes 扩散版本中得到了证明，显示于例 7-21 中。

例 7-21　ctypes 2 阶扩散代码

```
import ctypes

grid_shape = (512, 512)
_diffusion = ctypes.CDLL("../diffusion.so") # ❶

# Create references to the C types that we will need to simplify future code
TYPE_INT = ctypes.c_int
TYPE_DOUBLE = ctypes.c_double
TYPE_DOUBLE_SS = ctypes.POINTER(ctypes.POINTER(ctypes.c_double))

# Initialize the signature of the evolve function to:
# void evolve(int, int, double**, double**, double, double)
_diffusion.evolve.argtypes = [
    TYPE_INT,
    TYPE_INT,
    TYPE_DOUBLE_SS,
    TYPE_DOUBLE_SS,
    TYPE_DOUBLE,
    TYPE_DOUBLE,
]
_diffusion.evolve.restype = None

def evolve(grid, out, dt, D=1.0):
    # First we convert the Python types into the relevant C types
```

① 这很依赖于 cPython。其他版本的 Python 可能有它们自己版本的 ctypes，可能以很不同的方式工作。

```
    cX = TYPE_INT(grid_shape[0])
    cY = TYPE_INT(grid_shape[1])
    cdt = TYPE_DOUBLE(dt)
    cD = TYPE_DOUBLE(D)
    pointer_grid = grid.ctypes.data_as(TYPE_DOUBLE_SS) # ❷
    pointer_out = out.ctypes.data_as(TYPE_DOUBLE_SS)

    # Now we can call the function
    _diffusion.evolve(cX, cY, pointer_grid, pointer_out, cD, cdt) # ❸
```

❶ 这类似于导入 diffusion.so 库。

❷ grid 和 out 都是 numpy 数组。

❸ 我们最终做完所有必要的步骤并能直接调用 C 函数。

我们做的第一件事就是“导入”我们的共享库。通过 ctypes.CDLL 调用来做。在这行中，我们可以声明 Python 能够访问的任何共享库（例如，ctypes-opencv 模块装载 libcv.so 库）。由此，我们得到了一个_diffusion 对象，包含了共享库所含有的所有成员。在这个例子中，diffusion.so 只包含了一个函数——evolve，这不是一个对象的属性。如果 diffusion.so 中含有许多函数和属性，我们能通过 _diffusion 对象来访问它们全部。

无论怎样，即使_diffusion 对象里有可调用的 evolve 函数，却不知道怎样来使用它。C 是静态类型的，并且函数有很具体的签名。为了恰当地用 evolve 函数来工作，我们必须显式地设置输入参数类型和返回类型。当用 Python 接口串行地开发库，或者当处理一个快速变化的库时，这就会变得很枯燥。而且，既然 ctypes 不能检查你所给的是否是正确的类型，一旦你犯错，你的代码就可能默默地失败或发生段错误。

而且除了要设置参数和函数对象的返回类型，我们也需要小心地转换使用的数据（这叫“类型转换”）。我们传送给函数的每一个参数必须很小心地转换为本地的 C 类型。有时这事会变得相当诡异，因为 Python 的变量类型很灵活。例如，如果我们有 num1 = 1e5，我们就知道这是一个 Python 的浮点数，因此我们应该使用 ctypes.c_float。另一方面，对于 num2 = 1e300 来说，我们就不得不使用 ctype.c_double，因为它对标准 C 浮点数会有溢出。

那就是说，numpy 给它的数组提供了一个.ctypes 属性来使它易与 ctypes 兼容。如果 numpy 没有提供这个功能，我们将不得不把一个 ctypes 数组初始化成正确类型，接着找到我们原始数据的位置并让我们的新 ctypes 对象指向那儿。

警告

当你正在把一个对象转换为一个 ctype 对象时，除非那个对象实现了一个缓存（就如 array 模块、numpy arrays、cStringIO 等那样），不然你的数据将会被拷贝进新对象中。在把一个整数转换为浮点数的情况下，这并不会对你的代码性能有什么影响。无论如何，如果你正在转换一个很长的 Python list，这会导致很大的性能惩罚！在这些情况下，使用 array 模块，或 numpy array，或者干脆用 struct 模块构建你自己的缓存对象，这些都会有帮助。但是这样做会伤害你的代码可读性，因为这些对象一般没有它们的原生 Python 对应物那样灵活。

如果你必须要给库传送一个复杂的数据结构，情况就会变得愈加复杂。例如，如果你的库期待一个代表空间中的点的 C 结构，有 x 和 y 属性，你不得不定义：

```
from ctypes import Structure
class cPoint(Structure):
    _fields_ = ("x", c_int), ("y", c_int)
```

在这个点上，你能通过初始化一个 cPoint 对象（例如，point = cPoint(10, 5)）来开始创建一个兼容 C 的对象。这不是一项有可怕工作量的工作，但是却会变得乏味，并且产生一些代码碎片。如果一个新版本库发布了，稍微改了下结构体，会发生什么事呢？这会让你的代码很难维护，一般会导致僵硬的代码，开发者会决定从不去更新正在被使用的低层库。

基于这些理由，如果你已经很好地理解 C 并且想要能够微调接口的每一方面，使用 ctypes 模块是很正确的。它有很好的可移植性，因为它是标准库的一部分，如果你的任务简单，它就提供简单的解决方案。要仔细一点，因为 ctypes 解决方案的复杂性（类似低层解决方案）会很快变得难以管理。

7.13.2 cffi

意识到 ctypes 有时会相当难用，cffi 意图简化程序员所使用的许多标准运算。它用一个内部的 C 解析器来做，能够理解函数和结构体定义。

作为结果，我们能仅仅写出 C 代码来定义我们希望使用的库的结构体，接着 cffi 将会为我们做所有重量级的工作：导入模块并确认我们给结果函数声明了正确的类型。事实上，如果有库的源码，这个工作几乎微不足道，因为头文件（用.h 结尾的文件）会包含我们所需的所有相关定义。例 7-22 演示了 cffi 版本的 2 阶扩散代码。

例 7-22　cffi 的 2 阶扩散代码

```
from cffi import FFI

ffi = FFI()
ffi.cdef(r'''
   void evolve(
    int Nx, int Ny,
    double **in, double **out,
    double D, double dt
   ); # ❶
''')
lib = ffi.dlopen("../diffusion.so")

def evolve(grid, dt, out, D=1.0):
    X, Y = grid_shape
    pointer_grid = ffi.cast('double**', grid.ctypes.data) # ❷
    pointer_out = ffi.cast('double**', out.ctypes.data)
    lib.evolve(X, Y, pointer_grid, pointer_out, D, dt)
```

❶ 这个定义的内容能被正常地从你正在使用的库手册中，或者通过查看库的头文件获取。

❷ 尽管我们还是需要把非本地 Python 对象做类型转换来使用我们的 C 模块，语法对那些有 C 经验的人来说是很熟悉的。

在前面的代码中，我们可以把 cffi 的初始化过程看作两个步骤。首先，我们创建了一个 FFI 对象并且给出我们所需的全局 C 声明。除了函数签名以外，还可以包括数据类型。接着，我们使用 dlopen 把一个共享库导入它自己的名字空间，这是一个 FFI 的子空间。这意味着我们能够装载两个具有相同 evolve 函数的库，分别赋给变量 lib1 和 lib2，然后独立地使用它们（这对于调试和剖析很给力）。

除了简单地导入 C 共享库以外，cffi 允许你只写 C 代码，然后使用 verify 函数来即时编译。这有很多即时收益——你能够简单地把你的一小部分代码用 C 来重写，而不去调用独立的 C 库的庞大的机制。可做替换的是，如果有一个你希望使用的库，但是要求用一些 C 的胶水代码来让接口完美工作，你可以按例 7-23 显示的那样只是把它和你的 cffi 代码内联起来从而让一切都处于一个中心化的位置上。除此之外，既然代码是即时编译的，你可以给你需要编译的每块代码声明编译指令。需要注意的是，无论如何，当每次 verify 函数运行去做实际的编译时，这种编译方式会有首次惩罚。

例 7-23　内联 2 阶扩散代码的 cffi

```
ffi = FFI()
ffi.cdef(r'''
```

```
    void evolve(
        int Nx, int Ny,
        double **in, double **out,
        double D, double dt
    );
''')
lib = ffi.verify(r'''
void evolve(int Nx, int Ny,
        double in[][Ny], double out[][Ny],
        double D, double dt) {
    int i, j;
    double laplacian;
    for (i=1; i<Nx-1; i++) {
        for (j=1; j<Ny-1; j++) {
            laplacian = in[i+1][j] + in[i-1][j] + in[i][j+1] + in[i][j-1]\
                - 4 * in[i][j];
            out[i][j] = in[i][j] + D * dt * laplacian;
        }
    }
}
''', extra_compile_args=["-O3",]) # ❶
```

❶ 既使我们正在即时编译这段代码，我们也可以提供相关的编译标志。

Verify 功能的另一个好处是它与复杂的 cdef 声明交互得很好。例如，如果我们正要使用一个具有很复杂结构体的库，但是只是想使用它的一部分，我们能够使用部分结构定义。为此，我们在 ffi.cdef 中的结构体定义中添加一个…，并且在后面的 verify 中#include 相关的头文件。

例如，假设我们正使用一个有 complicated.h 头文件的库，其中包含了一个类似于下面的结构体：

```
struct Point {
    double x;
    double y;
    bool isActive;
    char *id;
    int num_times_visited;
}
```

如果我们只需要关心 x 和 y 的属性，我们能写一些简单的 cffi 代码，只关心下面这些值：

```
from cffi import FFI

ffi = FFI()
ffi.cdef(r"""
    struct Point {
```

```
        double x;
        double y;
        ...;
    };
    struct Point do_calculation();
""")
lib = ffi.verify(r"""
    #include <complicated.h>
""")
```

我们能运行 comlicated.h 库的 do_calculation 函数，并且返回给我们一个 Point 对象，它的 x 和 y 属性是可以访问的。这种移植性很惊人，因为代码在具有不同的 Point 实现的系统上都能很好地工作，或者当 complicated.h 的新版本出来后也能很好地工作，只要它们都有 x 和 y 属性就可以。

所有这些优点让 cffi 成为在 Python 中用 C 代码来工作时的一个很优秀的工具。它比 ctypes 简单很多，然而当直接和外来函数接口一起用时，还是给予了你想要的等量的细粒度控制。

7.13.3 f2py

对于许多科学应用来说，Fortran 还是一个黄金标准。尽管它作为一种通用语言的日子已经结束了，但它还是有很多优点来让编写矢量运算变得容易，并且运行相当快。另外，有很多性能数学库用 Fortran 编写（LAPACK、BLAS 等），并且能够使用在你的 Python 代码中的性能关键部分。

对于这种情况，f2py 提供了一个非常简单的把 Fortran 代码导入 Python 的方法。归因于 Fortran 的显式类型，这个模块能做得很简单。既然解析和理解类型是简单的，f2py 就能容易地创建一个 CPython 模块，该模块依靠 C 中的本地外部函数支持来使用 Fortran 代码。这意味着当你要使用 f2py 时，你其实在自动生成一个知道怎样使用 Fortran 代码的 C 模块！这样的结果就是，许多在 ctypes 和 cffi 的方案中与生俱来的混淆就不存在了。

在例 7-24 中，我们能看到一些用于解决扩散方程的与 f2py 兼容的代码。事实上，所有本地 Fortran 代码都与 f2py 兼容。无论如何，函数参数注解（由!f2py 为前缀的语句）简化了最终的 Python 模块并且会生成易于使用的接口。注解隐式地告诉 f2py 我们是否意图让一个参数只用于输出还是只用于输入，是让我们就地修改还是完全隐式地修改。隐式类型对于 vectors 的大小尤其有用：当 Fortran 可能需要让那些数字变显式时，我们的 Python 代码已经有这个信息在手上了。当我们把类型设为“隐式”时，f2py 能够自动为我们填充那些值，基本上在最终的 Python 接口中是让它们保持隐式的。

例 7-24　使用 f2py 注解的 Fortran 2 阶扩散代码

```
SUBROUTINE evolve(grid, next_grid, D, dt, N, M)
    !f2py threadsafe
    !f2py intent(in) grid
    !f2py intent(inplace) next_grid
    !f2py intent(in) D
    !f2py intent(in) dt
    !f2py intent(hide) N
    !f2py intent(hide) M
    INTEGER :: N, M
    DOUBLE PRECISION, DIMENSION(N,M) :: grid, next_grid
    DOUBLE PRECISION, DIMENSION(N-2, M-2) :: laplacian
    DOUBLE PRECISION :: D, dt

    laplacian = grid(3:N, 2:M-1) + grid(1:N-2, 2:M-1) + &
                grid(2:N-1, 3:M) + grid(2:N-1, 1:M-2) - 4 * grid(2:N-1, 2:M-1)
     next_grid(2:N-1, 2:M-1) = grid(2:N-1, 2:M-1) + D * dt * laplacian
END SUBROUTINE evolve
```

我们运行下面的命令来把代码构建成一个 Python 模块：

```
$ f2py -c -m diffusion --fcompiler=gfortran --opt='-O3' diffusion.f90
```

这会创建一个能被直接导入 Python 的 diffusion.so 文件。

如果我们交互性地运行结果模块，我们能看到 `f2py` 带给我们的好处，多亏了我们的注解和 `f2py` 解析 Fortran 代码的能力：

```
In [1]: import diffusion

In [2]: diffusion?
Type:         module
String form: <module 'diffusion' from 'diffusion.so'>
File:         .../examples/compilation/f2py/diffusion.so
Docstring:
This module 'diffusion' is auto-generated with f2py (version:2).
Functions:
  evolve(grid,next_grid,d,dt)
.

In [3]: diffusion.evolve?
Type:         fortran
String form: <fortran object>
Docstring:
evolve(grid,next_grid,d,dt)

Wrapper for ``evolve``.
```

```
Parameters
grid : input rank-2 array('d') with bounds (n,m)
next_grid : rank-2 array('d') with bounds (n,m)
d : input float
dt : input float
```

这个代码显示出 f2py 生成的结果是自动文档化的，而且接口相当简化。例如，与其我们来抽取出 vectors 的大小，f2py 已经能发现自动寻找这些信息的方法并且在最终的接口中只是把它隐藏了起来。事实上，最终的 evolve 函数签名看起来和我们在例 6-14 中所写的纯 Python 版本一模一样。

我们必须要仔细的唯一事情就是在内存中 numpy array 的顺序。既然绝大多数我们所用的 numpy 和 Python 集中于从 C 演变过来的代码，我们对于内存中的数据顺序一直使用 C 的惯例（称为行优先顺序）。Fortran 使用了一种不同的惯例（列优先顺序），这样我们必须要确保让我们的 vectors 遵守惯例。这些顺序仅仅表明了对于一个 2 维数组，它的列或行在内存中是否是紧挨的[①]。幸运的是，这仅仅意味着当我们声明 vectors 时，给 numpy 指明 order='F' 参数就行。

备忘

行优先顺序和列优先顺序的差异意味着矩阵[[1, 2], [3, 4]]在内存中按行优先顺序排列成[1, 2, 3, 4]，按列优先顺序排列成[1, 3, 2, 4]。这个差异只是习惯上的，当使用恰当时对性能没有任何真正影响。

这就产生了下面我们使用 Fortran 子例程的代码。除了导入 f2py 衍生库和让我们的数据遵照显式的 Fortran 顺序之外，这个代码看上去和我们在例 6-14 中所使用的简直一模一样。

```
from diffusion import evolve

def run_experiment(num_iterations):
    next_grid = np.zeros(grid_shape, dtype=np.double, order='F') # ❶
    grid = np.zeros(grid_shape, dtype=np.double, order='F')

    # ... standard initialization ...

    for i in range(num_iterations):
        evolve(grid, next_grid, 1.0, 0.1)
        grid, next_grid = next_grid, grid
```

① 更多信息，请参看维基。

❶ Fortran 在内存中以不同的顺序排列数字，所以我们必须记得把我们的 `numpy array` 设置成使用这个标准。

7.13.4 CPython 模块

最后，我们总是能一路走到 CPython API 的层面，并且写一个 CPython 模块。这要求我们以和开发 CPython 一样的方式去写代码，并且需要小心对待我们的代码和 CPython 实现之间的所有交互。

这样会具有很优秀的移植性优势，取决于 Python 版本。我们不需要任何外部模块或库，只需一个 C 编译器和 Python！无论怎样，这样并不一定能很好地扩展到 Python 的新版本中去。例如，让用 Python 2.7 写的 CPython 模块和 Python 3 一起工作。

然而为移植性付出了巨大的代价，你要负责你的 Python 代码和模块之间接口的所有方方面面。甚至只是为最简单的任务也要去写上几十行的代码。例如，为了和来自例 7-20 中的扩散库对接，我们必须写 28 行代码只是去读一个函数的参数并解析它（例 7-25）。当然，这确实意味着你有惊人的细粒度去控制正在发生的事情。如此可以一直走下去直到能够手动为 Python 的垃圾收集去更改引用计数（当创建处理本地 Python 类型的 CPython 模块时，这是许多痛楚的根源）。就因为这样，最终的代码趋向于比其他的接口方式快几分钟。

警告

总而言之，这个方法应该留作最后的解决手段。虽然它为编写 CPython 模块提供了很多信息，但最终的代码不如其他潜在的方法那么可重用和可维护。在模块中做很细微的改动常常需要完全重写。事实上，我们包含了模块代码，并且需要 setup.py 来编译它（例 7-26）作为警示。

例 7-25　与 2 阶扩散库对接的 CPython 模块

```
// python_interface.c
// - cpython module interface for diffusion.c
#define NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION

#include <Python.h>
#include <numpy/arrayobject.h>
#include "diffusion.h"

/* Docstrings */
```

```
static char module_docstring[] =
    "Provides optimized method to solve the diffusion equation";
static char cdiffusion_evolve_docstring[] =
    "Evolve a 2D grid using the diffusion equation";

PyArrayObject* py_evolve(PyObject* self, PyObject* args) {
    PyArrayObject* data;
    PyArrayObject* next_grid;
    double dt, D=1.0;

    /* The "evolve" function will have the signature:
    *       evolve(data, next_grid, dt, D=1)
    */
    if (!PyArg_ParseTuple(args, "OOd|d", &data, &next_grid, &dt, &D)) {
        PyErr_SetString(PyExc_RuntimeError, "Invalid arguments");
        return NULL;
    }

    /* Make sure that the numpy arrays are contiguous in memory */
    if (!PyArray_Check(data) || !PyArray_ISCONTIGUOUS(data)) {
        PyErr_SetString(PyExc_RuntimeError,"data is not a contiguous array.");
        return NULL;
    }
    if (!PyArray_Check(next_grid) || !PyArray_ISCONTIGUOUS(next_grid)) {
        PyErr_SetString(PyExc_RuntimeError,"next_grid is not a contiguous array.");
        return NULL;
    }

    /* Make sure that grid and next_grid are of the same type and have the same
    * dimensions
    */
    if (PyArray_TYPE(data) != PyArray_TYPE(next_grid)) {
        PyErr_SetString(PyExc_RuntimeError,
                        "next_grid and data should have same type.");
        return NULL;
    }
    if (PyArray_NDIM(data) != 2) {
        PyErr_SetString(PyExc_RuntimeError,"data should be two dimensional");
        return NULL;
    }
    if (PyArray_NDIM(next_grid) != 2) {
        PyErr_SetString(PyExc_RuntimeError,"next_grid should be two dimensional");
        return NULL;
    }
```

```
    if ((PyArray_DIM(data,0) != PyArrayDim(next_grid,0)) ||
        (PyArray_DIM(data,1) != PyArrayDim(next_grid,1))) {
        PyErr_SetString(PyExc_RuntimeError,
                        "data and next_grid must have the same dimensions");
        return NULL;
    }

    /* Fetch the size of the grid we are working with */
    const int N = (int) PyArray_DIM(data, 0);
    const int M = (int) PyArray_DIM(data, 1);

    evolve(
        N,
        M,
        PyArray_DATA(data),
        PyArray_DATA(next_grid),
        D,
        dt
    );

    Py_XINCREF(next_grid);
    return next_grid;
}

/* Module specification */
static PyMethodDef module_methods[] = {
/* { method name , C function , argument types , docstring } */
   { "evolve"     , py_evolve  , METH_VARARGS    , cdiffusion_evolve_docstring } ,
   { NULL         , NULL       , 0               , NULL }
};

/* Initialize the module */
PyMODINIT_FUNC initcdiffusion(void)
{
    PyObject *m = Py_InitModule3("cdiffusion", module_methods, module_docstring);
    if (m == NULL)
        return;
    /* Load `numpy` functionality. */
    import_array();
}
```

为了构建这个代码，我们需要创建一个 setup.py 脚本，使用 distutils 模块来确定构建代码的方式，这样才是兼容 Python（例 7-26）。除了标准 distutils 模块，numpy 还提供了它自己的模块用以帮助在你的 CPython 模块中加入与 numpy 的整合。

例 7-26　用于 CPython 模块扩散接口的 setup 文件

```
"""
setup.py for cpython diffusion module. The extension can be built by running

    $ python setup.py build_ext --inplace

which will create the __cdiffusion.so__ file, which can be directly imported into
Python.
"""

from distutils.core import setup, Extension
import numpy.distutils.misc_util

__version__ = "0.1"

cdiffusion = Extension(
    'cdiffusion',
    sources = ['cdiffusion/cdiffusion.c', 'cdiffusion/python_interface.c'],
    extra_compile_args = ["-O3", "-std=c99", "-Wall", "-p", "-pg", ],
    extra_link_args = ["-lc"],
)

setup (
    name = 'diffusion',
    version = __version__,
    ext_modules = [cdiffusion,],
    packages = ["diffusion", ],
    include_dirs = numpy.distutils.misc_util.get_numpy_include_dirs(),
)
```

生成的结果是一个 `cdiffusion.so` 文件，能被 Python 直接导入，而且使用起来相当简单。既然我们已经完全控制了最终函数的签名以及我们的 C 代码与库的精确交互方式，我们能够（以一些艰难的工作）创建出一个容易使用的模块。

```
from cdiffusion import evolve

def run_experiment(num_iterations):
    next_grid = np.zeros(grid_shape, dtype=np.double)
    grid = np.zeros(grid_shape, dtype=np.double)

    # ... standard initialization ...

    for i in range(num_iterations):
        evolve(grid, next_grid, 1.0, 0.1)
        grid, next_grid = next_grid, grid
```

7.14 小结

本章所介绍的各种不同的策略允许你在不同程度上定制你的代码，以便降低 CPU 必须执行的指令数量和提高你的程序效率。有时这能用算法来做到，尽管常常必须手动去做（请看 7.2 节）。而且，有时只是必须借用这些方法来使用其他语言已经写好的库。无论动机如何，Python 允许我们从其他语言在某些问题上能够带来的速度提升中获得收益，然而当有需要时还是保留了它的表达性和灵活性。

然而需要注意的是，做这些优化只是为了优化 CPU 指令的效率。如果你把 I/O 密集型进程和 CPU 密集型问题耦合了起来，仅仅是编译你的代码可能不会带来任何合理的速度提升。对于这些问题，我们必须重新思考我们的解决方案，并且可能要利用并行化来在同一时间运行不同的任务。

第 8 章

并发

读完本章之后你将能够回答下列问题

- 什么是并发，它如何起帮助作用？
- 并发和并行的区别是什么？
- 什么任务能够用并发来做，什么不能做？
- 并发的各种模式是什么？
- 什么时候是利用并发的合适时机？
- 并发如何来加速我们的程序？

I/O 对程序的执行流是相当大的负担。每一次你的代码读取一个文件或者写入一个网络 socket，它必须得暂停和内核的联系，请求去启动操作，并等待它完成。这可能看起来并不像是世界末日，尤其当你意识到每次分配内存时只会发生一次简单的操作之后。无论如何，如果我们回溯到图 1-3，就会看到我们执行的绝大多数 I/O 操作在比 CPU 慢几个数量级的设备上。

例如，一个典型操作大约花费 1 毫秒来写网络 socket，在这期间，我们本应能在一台 2.4GHz 的电脑上完成 24000000 条指令。最糟的是，我们的程序暂停超过了 1 毫秒——我们的执行流暂停下来了，我们正在等待一个写操作完成的信号。在暂停的状态下花费的时间叫作“I/O 等待”。

并发允许我们在等待一个 I/O 操作完成的时候执行其他操作，从而帮助我们把这个浪费的时间利用起来。例如，在图 8-1 中，我们看到它描述了一个必须运行三个任

务的程序，所有任务都在其内具有周期性的 I/O 等待。如果我们串行地运行它们，我们就会遭受三次 I/O 等待的惩罚。

无论如何，如果我们并发地运行这些任务，我们基本上就能通过同时运行其他的任务来隐藏掉等待的时间。值得注意的是，这还是都发生在一个单独的线程上，还是每次只使用一个 CPU！

尽管并发不局限于 I/O，但这是我们所见到的能得到最大收益的地方。在一个并发程序中，与其让你的代码串行执行——那就是，从一行到下一行——不如编写你的代码来处理事件，当不同事件发生时，让你的代码运行于不同的部分。

通过用这种方式对一个程序建模，我们就能够处理我们所关心的特殊事件：I/O 等待。

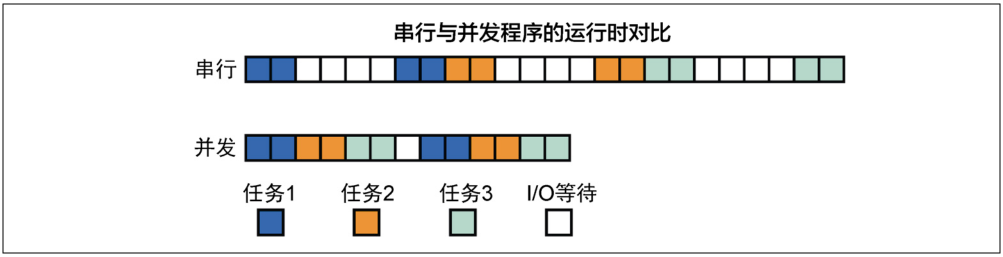


图 8-1　串行和并发程序的对比

8.1　异步编程介绍

当一个程序进入 I/O 等待时，暂停执行，这样内核就能执行 I/O 请求相关的低级操作（这叫作一次上下文切换），直到 I/O 操作完成时才继续。上下文切换是相当重量级的操作。它要求我们保存程序的状态（丢失了我们在 CPU 层面上任何类型的缓存），放弃使用 CPU。之后，当我们允许再次运行时，我们必须花时间在主板上重新初始化程序并准备好继续运行（当然，所有这一切都在幕后发生）。

另一方面，使用并发，典型情况下我们会有一个叫作“事件循环”的东西，来管理我们程序中该运行什么，什么时候运行。实质上，一个事件循环只是需要运行的一个函数列表。在列表顶端的函数得到运行，接着轮到下一个，依次类推。例 8-1 展示了一个事件循环的简单例子。

例 8-1　一个玩具意义上的事件循环

```
from Queue import Queue
from functools import partial

eventloop = None
```

```
class EventLoop(Queue):
    def start(self):
        while True:
            function = self.get()
            function()

def do_hello():
    global eventloop
    print "Hello"
    eventloop.put(do_world)

def do_world():
    global eventloop
    print "world"
    eventloop.put(do_hello)

if __name__ == "__main__":
    eventloop = EventLoop()
    eventloop.put(do_hello)
    eventloop.start()
```

这可能看上去不像一个巨大的变化。无论如何，当运行 I/O 任务时，我们能够耦合事件循环和异步（async）I/O 操作来获得巨大的收益。这些操作是非阻塞的，意味着如果我们用一个异步程序做一次网络写操作，它会立即返回，尽管写操作还没有发生。当写操作完成时，会触发一个事件，所以我们的程序就得以知晓了。

把这两个概念放在一起，我们就能获得这样一个程序：当请求一个 I/O 操作时，在等待原来的 I/O 操作完成期间，可以运行另外的函数。这实质上还允许我们做有意义的运算，不然我们就会处于 I/O 等待中。

备忘

函数之间的切换会有开销。内核必须花费时间来设置在内存中被调用的函数，我们缓存的状态将会变得不可预测。正因为如此，在你的程序有许多 I/O 等待时，并发给出了最好的结果——尽管这种切换的确有它的开销，但它要比把 I/O 等待时间利用起来从而取得的收益要小得多。

使用事件循环编程能采取两种方式：回调或者 future。在回调模式中，使用一个通常称之为回调的函数作为输入参数来调用函数。它会使用值来调用回调函数，而不是把值返回出去。这样就设置了长长的调用函数链，每一个函数得到链中前一个函数返回的值。例 8-2 是一个回调模式的简单例子。

例 8-2　回调例子

```
from functools import partial
def save_value(value, callback):
    print "Saving {} to database".format(value)
    save_result_to_db(result, callback) # ❶

def print_response(db_response):
    print "Response from database: {}".format(db_response)

if __name__ == "__main__":
    eventloop.put(
        partial(save_value, "Hello World", print_response)
    )
```

❶ `save_result_to_db` 是一个异步函数，它会立即返回并且结束，允许其他代码运行。无论如何，一旦数据准备好，`print_response` 就会被调用。

另一方面，使用 futures，一个异步函数返回一个 future 结果的 promise，而不是实际的结果。正因为如此，我们必须等待被这种类型的异步函数所返回的 future 完成，并被我们所期待的值所填充（或者在其中做一个 `yield`，或者通过运行一个函数显式地等待值准备好）。在等待 future 对象被我们所请求的数据填充时，我们能够做其他运算。如果我们把它和生成器（generators）的概念——能够被暂停并且以后能继续执行的函数——耦合起来，我们就能写出看上去形式上很接近串行代码的异步代码：

```
@coroutine
def save_value(value, callback):
    print "Saving {} to database".format(value)
    db_response = yield save_result_to_db(result, callback) # ❶
    print "Response from database: {}".format(db_response)

if __name__ == "__main__":
    eventloop.put(
        partial(save_value, "Hello World")
    )
```

❶ 在这种情况下，`save_result_to_db` 返回一个 `Future` 类型。通过让步（yielding），我们就确保暂停了 `save_value`，直到值准备好了才继续并完成它的操作。

在 Python 中，协程是作为生成器（generator）来实现的。这很方便，因为生成器（generator）已经有机制来暂停它们的执行并在以后继续运行。所以，在我们的协程中所发生的事情就是产生一个 future，事件循环会等待直到那个 future 把它的值准备好。一旦值准备好了，事件循环会继续执行那个函数，把 future 的值送还给它。

对于 Python 2.7 的基于 future 的并发实现，当我们设法把协程用作实际函数时，事情会变得有点奇怪。记住生成器（generators）不能返回值，所以就有了库处理这个问题的各种各样的方式。

在 Python 3.4 中，无论如何，已经引入了新的机制来方便地创建协程，并且还是让协程来返回值。

在本章中，我们将分析一个从 HTTP 服务器抓取数据的网络爬虫，这个 HTTP 服务器被构造成有延迟。这代表了无论何时当我们处理 I/O 时会发生的普遍的响应时间延迟。我们首先创建一个串行爬虫，看起来就像天然的针对这个问题的 Python 解决方案。接着，我们浏览 Python 2.7：gevent 和 tornado 这两种解决方案。最后，我们会查看 Python 3.4 中的 asyncio 库，看看在 Python 中异步编程的未来会是什么样的。

备忘

我们实现的 Web 服务器能够同时支持多个连接。这对于你要运行 I/O 操作所遇到的大多数的服务来说是真实的 —— 大多数数据库能够同时支持多个请求，大多数 Web 服务器支持 10000 多个同时连接。无论如何，当与一个不能同时处理多个连接的服务交互时[①]，我们将总是获得与串行情况下相同的性能。

8.2 串行爬虫

对于在我们实验中的并发控制，我们会写一个串行的 Web 爬虫来使用一串 URL 列表，抓取它们并对页面内容的总长度求和。我们会使用一个定制的 HTTP 服务器，采用两个参数，name 和 delay。delay 域会告诉服务器在得到响应前暂停多久，以毫秒计时。name 域只是为了日志的目的。

通过控制 delay 参数，我们能够模拟服务器响应我们请求的时间。在真实的世界中，这可能对应于一个慢速的 Web 服务器，一个繁重的数据库调用，或是任何要花长时间运行的 I/O 调用。对于串行的情况，这仅仅代表了我们的程序会陷入更多的 I/O 等待的时间，但是在之后的并发例子中，它也代表了程序能花更多的时间来做其他事情。

此外，我们选择使用 requests 模块来执行 HTTP 调用。做出这个选择是出于这个模块的简单性。我们一般为此小节使用 HTTP，因为它是一个简单的 I/O 例子，

① 对于一些数据库，比如 Redis，这是一个专为维护数据一致性所做出的设计抉择。

并且能相当容易地执行 HTTP 请求。一般情况下，对一个 HTTP 库的任何调用都能被其他 I/O 所代替。我们串行版本的 HTTP 爬虫显示于例 8-3 中。

例 8-3　串行 HTTP 爬虫

```
import requests
import string
import random

def generate_urls(base_url, num_urls):
    """
    We add random characters to the end of the URL to break any caching
    mechanisms in the requests library or the server
    """

    for i in xrange(num_urls):
        yield base_url + "".join(random.sample(string.ascii_lowercase, 10))

def run_experiment(base_url, num_iter=500):
    response_size = 0
    for url in generate_urls(base_url, num_iter):
        response = requests.get(url)
        response_size += len(response.text)
    return response_size

if __name__ == "__main__":
    import time
    delay = 100
    num_iter = 500
    base_url = "http://127.0.0.1:8080/add?name=serial&delay={}&".format(delay)

    start = time.time()
    result = run_experiment(base_url, num_iter)
    end = time.time()
    print("Result: {}, Time: {}".format(result, end - start))
```

当运行这段代码时，会看到一个有趣的度量就是去测试 HTTP 服务器所见到的每个请求的起始和结束时间。这告诉我们在 I/O 等待期间的代码效率——因为我们的任务只是去发起 HTTP 请求并对返回的字节数求和，我们应该能够在等待其他请求完成的期间，发起更多的 HTTP 请求，并处理任何响应。

从图 8-2 中，我们可以看到，就如所期望的那样，我们的请求没有交织。我们在一个时间做一次请求，并且在我们转移到下一次请求之前，等待前面的请求做完。事实上，串行过程的整体运行时间很有意义，知道了这个：既然每个请求花费 0.1 秒（因为我们的 `delay` 参数），我们正在做 500 次请求，那么我们就期待整体运行时间大概是 50 秒。

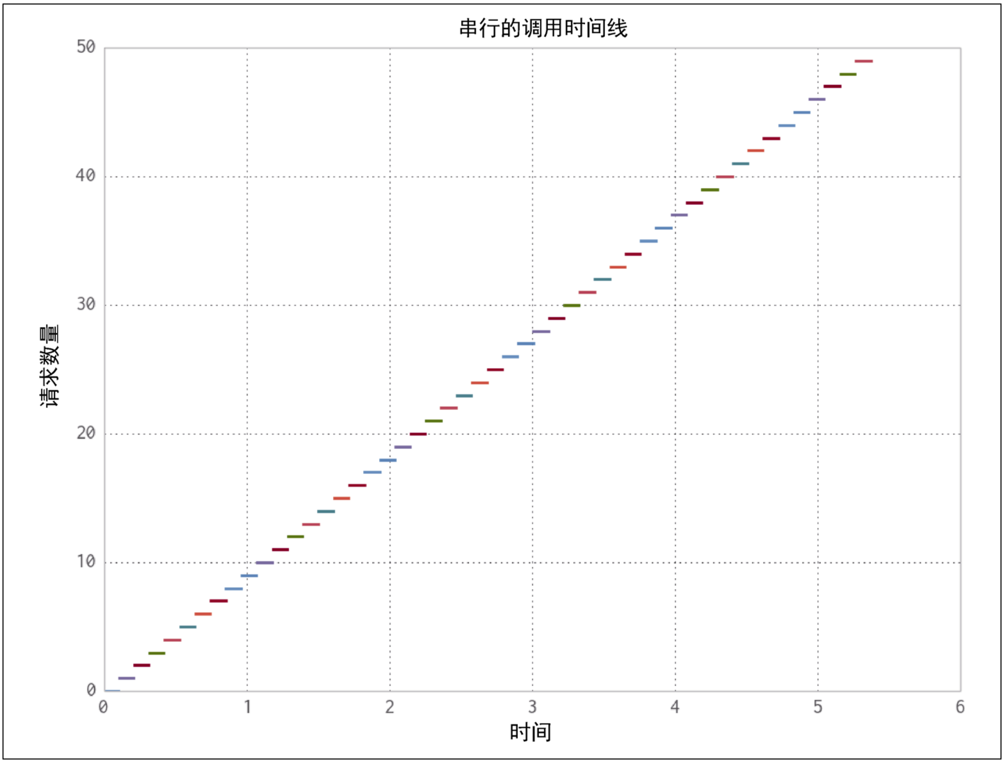


图 8-2　例 8-3 的 HTTP 请求时间表

8.3　gevent

最简单的异步库之一就是 gevent。它遵照了让异步函数返回 future 的模式，意味着代码中的大部分逻辑会保持一样。此外，gevent 对标准 I/O 函数做了猴子补丁，把它们变成了异步，这样大多数时间你可以仅仅使用标准的 I/O 包并得益于异步的行为。

gevent 提供了两个机制来使能异步编程——就如我们刚才提到的，它用异步的 I/O 函数给标准库打补丁，并且它也有一个 greenlet 对象能被用于并发执行。greenlet 是一种协程，能够被想象成线程（请看第 9 章对线程的讨论）。无论怎样，所有的 greenlets 在同一物理线程上运行。那就是说，gevent 的调度器在 I/O 等待期间使用一个事件循环在所有 greenlets 间来回切换，而不是用多个 CPU 来运行它们。大多数情况下，gevent 通过使用 wait 函数来设法尽可能透明化地处理事件循环。wait 函数将启动一个事件循环，只要有需要就运行着，直到所有的 greenlets 结束。正因如此，你的大部分 gevent 代码以

串行方式运行。接着，在某点上，你会设置许多 `greenlets` 来做并发任务，并且用 `wait` 函数来启动事件循环。当 `wait` 函数正在执行时，你入队堆积起来的所有并发任务会运行直到结束（或某个停止条件），接着你的代码会重新回到串行方式运行。

Future 由 `gevent.spawn` 来创建，使用了一个函数和传递给这个函数的参数，并且启动了一个负责运行这个函数的 `greenlet`。greenlet 能够被看作一个 future，因为你声明的函数一旦运行完成，它的值就会包含在 greenlet 的 `value` 域中。

Python 标准模型的补丁会让人更难以控制异步函数得以运行的细节和时间。例如，当正在做异步 I/O 时，我们想要确认的一件事就是我们没有同时打开太多的文件或者连接。如果我们这样做了，就会让远程服务器过载，或者不得不在太多的操作间做上下文切换，从而减慢进程速度。启动与我们要抓取的 URL 相同数量的 greenlets 是没有效率的，我们需要一种机制来限制我们同时处理的 HTTP 请求。

我们能够通过使用信号量来手动控制并发请求的数量，从而同一时刻只从 100 个 greenlets 来做 HTTP 的 get 请求。信号量确保了同一时刻只有一定数量的协程能进入上下文模块。作为结果，我们能够启动我们所需的所有 greenlets 来立即抓取 URLs，但只有其中 100 个将会在同一时刻做出 HTTP 调用。信号量是一种在各种各样的并行代码流程中使用很多的加锁机制。通过基于各种不同的规则来限制你的代码进程，锁能够帮助你确保程序中各个不同的模块之间不会相互干扰。

现在既然我们设置好了所有的 futures，并且已经把加锁机制置入了 greenlets 的控制流，我们就能够等待直到用 `gevent.iwait` 函数开始获取结果为止，那样就会得到一个 futures 的序列，并遍历准备好的项。反之，我们本可以使用 `gevent.wait`，那会阻塞我们程序的执行直到所有的请求做完为止。

我们经历了把我们的请求分块化的麻烦，而不是把它们立即全都发送出去，因为超载的事件循环会导致性能降低（这对于所有的异步编程都是真实存在的）。从实验中，我们通常看到在同一时刻 100 个左右的打开连接是有优化作用的（见图 8-3）。如果我们打开更少的连接，我们就还是会在 I/O 等待期间浪费时间。如果打开更多的连接，我们就会在事件循环中太频繁地做上下文切换，给我们的程序增加不必要的负担。那就是说，100 这个值取决于许多事情——代码正运行于其上的计算机，事件循环的实现，远端主机的属性，远端主机的期望响应时间等。我们建议在决定选择之前做一些实验。例 8-4 显示了我们 HTTP 爬虫的 gevent 版本代码。

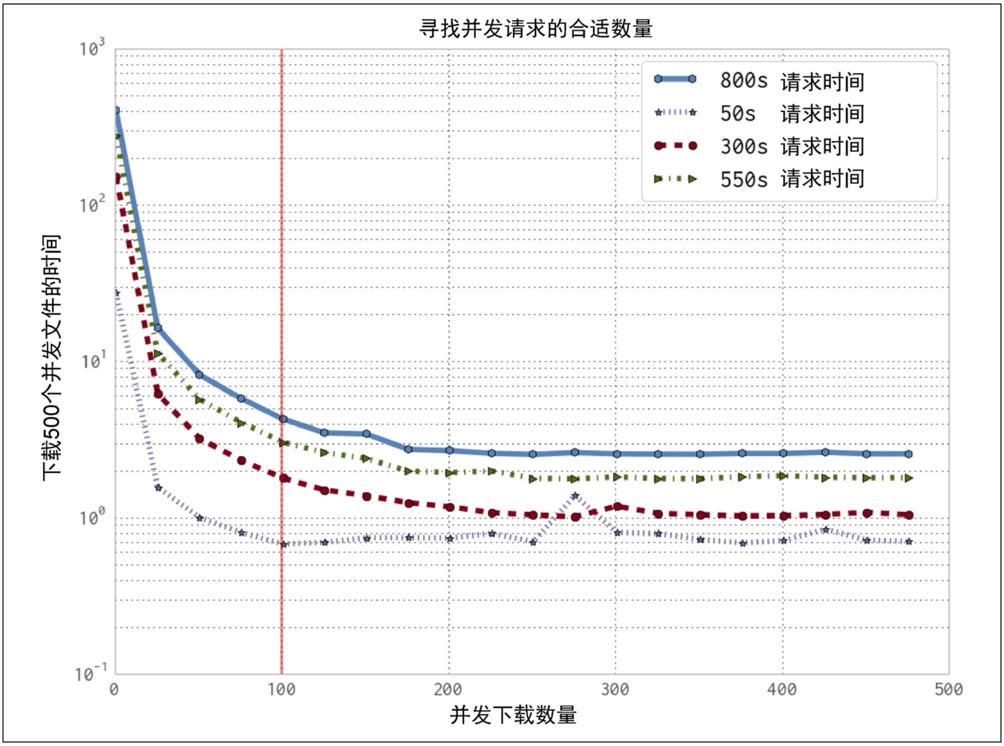


图 8-3 寻找并发请求的合适数量

例 8-4 gevent HTTP 爬虫

```
from gevent import monkey
monkey.patch_socket()

import gevent
from gevent.coros import Semaphore
import urllib2
import string
import random

def generate_urls(base_url, num_urls):
    for i in xrange(num_urls):
        yield base_url + "".join(random.sample(string.ascii_lowercase, 10))

def chunked_requests(urls, chunk_size=100):
    semaphore = Semaphore(chunk_size)  # ❶
    requests = [gevent.spawn(download, u, semaphore) for u in urls] # ❷
    for response in gevent.iwait(requests):
        yield response

def download(url, semaphore):
```

```
        with semaphore: # ❸
            data = urllib2.urlopen(url)
            return data.read()

def run_experiment(base_url, num_iter=500):
    urls = generate_urls(base_url, num_iter)
    response_futures = chunked_requests(urls, 100) # ❹
    response_size = sum(len(r.value) for r in response_futures)
    return response_size

if __name__ == "__main__":
    import time
    delay = 100
    num_iter = 500
    base_url = "http://127.0.0.1:8080/add?name=gevent&delay={}&".format(delay)

    start = time.time()
    result = run_experiment(base_url, num_iter)
    end = time.time()
    print("Result: {}, Time: {}".format(result, end - start))
```

❶ 这里我们生成了一个信号量来让 `chunk_size` 下载发生。

❷ 通过把信号量用作一个上下文管理器，我们确保了只有 `chunk_size` 数量的 greenlets 能够在同一时刻运行上下文的主体部分。

❸ 我们能够把所需数量的 greenlets 放在队列中，知道它们之中没有一个会运行直到我们用 `wait` 或 `iwait` 启动一个事件循环为止。

❹ `response_futures` 现在持有一个处于完成状态的 futures 的迭代器，所有这些 futures 的 `.value` 属性中都具有我们所期望的数据。

作为替换，我们能够使用 `grequests` 来大大简化我们的 `gevent` 代码。尽管 `gevent` 提供了所有类型的低级并发 `socket` 操作，`grequests` 组合了 HTTP 库请求和 `gevent`，其结果就是具有很简单的 API 来做并发 HTTP 请求（甚至为我们处理信号量逻辑）。使用 `grequests`，我们的代码变得简单很多，更容易理解，可维护性更好，然而却还是获得了与更低层的 `gevent` 代码相提并论的速度提升（见例 8-5）。

例 8-5　grequests HTTP 爬虫

```
import grequests

def run_experiment(base_url, num_iter=500):
    urls = generate_urls(base_url, num_iter)
    response_futures = (grequests.get(u) for u in urls) # ❶
```

```
    responses = grequests.imap(response_futures, size = 100) # ❷
    response_size = sum(len(r.text) for r in responses)
    return response_size
```

❶ 首先我们创建了请求并得到 future。我们选择了把它当作生成器（generator）来做，这样以后我们只需要做与我们准备发出的请求相同次数的估算。

❷ 现在我们能够取得 future 对象，并把它们映射成真实的响应对象。.imap 函数给我们一个生成器（generator）来产生响应对象，我们就是从响应对象来获取数据。

一件重要的事情要引起注意，那就是我们已经使用了 gevent 和 grequests 来生成异步的 I/O 请求，但是我们在 I/O 等待期间没有做任何非 I/O 的计算。图 8-4 显示了我们取得的巨大速度提升。通过在等待前面的请求完成之际发起更多的请求，我们能够取得 69 倍的速度提升！通过用水平线代表相互之间的请求栈的方式， 我们能够明显看到在前面的请求完成之前新的请求如何正在发送出去。这与串行爬虫（图 8-2）的例子形成了鲜明的对比，在串行爬虫的图中，一个线条只有在前面的线条完成之时才开始。而且，我们能够看到更有趣的效果伴随着 gevent 请求时间线的形状。例如，在大约前 100 个请求处，我们看到一个停顿，没有发起新请求。这是因为这时我们的信号量第一次命中，我们能够在任何前面的请求完成之前给信号量加锁。在这之后，信号量进入一个平衡态，只有当另一个请求完成时才解锁，并给当前新请求加锁。

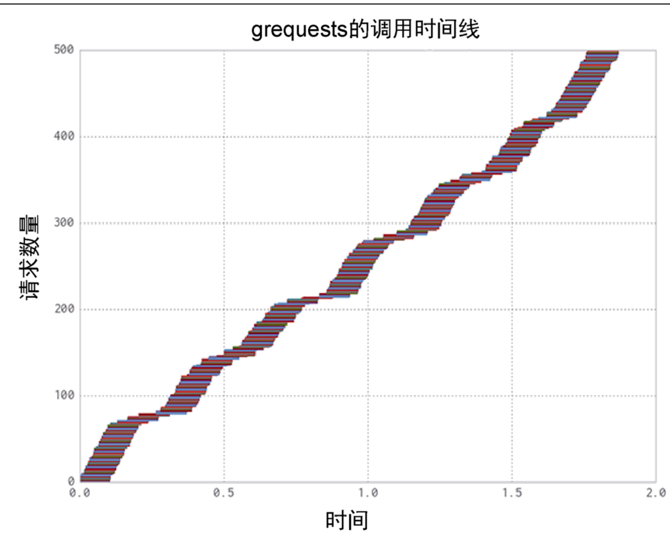


图 8-4　例 8-5 的 HTTP 请求时间表

8.4 tornado

另一个在 Python 中很频繁使用的异步 I/O 包是 `tornado`，由 Facebook 主要为 HTTP 客户端和服务器端开发。对比 `gevent`，`tornado` 选择使用回调的方式来做异步行为。无论怎样，在 3.x 发布版中，类似于协程的行为以一种和老代码兼容的方式添加了进来。

在例 8-6 中，我们实现了和用 `gevent` 相同的网络爬虫，但是使用了 `tornado` 的 I/O 循环（tornado 版的事件循环）和 HTTP 客户端。这样省却了我们的麻烦，比如不得不批量化我们的请求并处理其他更多底层代码的方方面面。

例 8-6　tornado HTTP 爬虫

```
from tornado import ioloop
from tornado.httpclient import AsyncHTTPClient
from tornado import gen

from functools import partial
import string
import random

AsyncHTTPClient.configure("tornado.curl_httpclient.CurlAsyncHTTPClient",
max_clients=100) # ❶

def generate_urls(base_url, num_urls):
    for i in xrange(num_urls):
        yield base_url + "".join(random.sample(string.ascii_lowercase, 10))

@gen.coroutine
def run_experiment(base_url, num_iter=500):
    http_client = AsyncHTTPClient()
    urls = generate_urls(base_url, num_iter)
    responses = yield [http_client.fetch(url) for url in urls] # ❷
    response_sum = sum(len(r.body) for r in responses)
    raise gen.Return(value=response_sum) # ❸

if __name__ == "__main__":
    #... initialization ...

    _ioloop = ioloop.IOLoop.instance()
    run_func = partial(run_experiment, base_url, num_iter)
    result = _ioloop.run_sync(run_func) # ❹
```

❶ 我们可以配置 HTTP 客户端并挑选我们希望使用的后台库，以及我们想要批量处理的请求数量。

❷ 我们生成了许多 futures，接着 yield 回到 I/O 循环中。这个函数会继续，`responses` 变量会被所有的 futures 填充，当它们就绪时，产生结果。

❸ 在 `tornado` 中的协程由 Python 的产生器（generators）来支持。为了从它们返回值，我们必须要生成一个特殊的异常，由 `gen.coroutine` 把它转化成一个返回值。

❹ `ioloop.run_sync` 会只在特殊化的函数`.ioloop.start()`的运行时间段内启动 IOLoop，另一方面，启动了一个必须手动停止的 IOLoop。

例 8-6 的 `tornado` 代码和例 8-4 的 `gevent` 代码的一个重要差别是在事件循环运行的时候。对于 `gevent` 来说，事件循环只有在 `iwait` 函数正运行的时候才运行。另一方面，在 `tornado` 中，事件循环在整个时间里都运行，并且控制着程序的完全执行流，而不仅仅是异步的 I/O 部分。

这使得 `tornado` 对于主要是 I/O 密集型，并且大部分程序（如果不是所有程序）是异步的应用来说很理想。`tornado` 所宣称的最大名声就是作为一个高性能的 web 服务器。事实上，Micha 已经编写了基于 `tornado` 的数据库和在很多场合需要许多 I/O 的数据结构[①]。在另一方面，既然 `gevent` 对你的整体程序没有要求，它对于主要是基于 CPU，然而有时需要重量级 I/O 的问题是一个理想的解决方案——例如，一个程序在一个数据集上做了很多计算，接着必须把结果送回数据库来存储。这样甚至会变得更简单，因为事实上大多数数据库有简单的 HTTP API，意味着你能够使用 `grequests`。

如果我们看看例 8-7 中更老风格的使用了回调的 `tornado` 代码，我们就能发现 `tornado` 的事件循环多么有控制力。我们可以看到为了启动代码，我们必须给程序添加入口点到 I/O 循环中，接着再启动它。然后，为了让程序终止，我们必须小心翼翼地给我们的 I/O 循环带上 `stop` 函数，并且在合适的时候调用它。结果就是，必须显式地带上回调的程序变得负担相当沉重，而且很快就变得不可维护了。这种情况发生的一个原因是回溯不能再持有变量信息，这些信息就是关于哪些函数调用了哪些函数，并且我们怎样进入到一个异常来启动。即使只是要完全知道调用了哪些函数也变得困难，因为我们一直在创建偏函数来填充参数。毫不奇怪，这就是通常所称的“回调地狱”。

例 8-7　使用回调的 tornado 爬虫

```
from tornado import ioloop
from tornado.httpclient import AsyncHTTPClient
```

① 例如，`fuggetaboutit` 是一种特殊类型的概率数据结构（请看 11.6 节）来使用 `tornado IOLoop` 调度时间任务。

```
from functools import partial

AsyncHTTPClient.configure("tornado.curl_httpclient.CurlAsyncHTTPClient",
                          max_clients=100)

def fetch_urls(urls, callback):
    http_client = AsyncHTTPClient()
    urls = list(urls)
    responses = []
    def _finish_fetch_urls(result): # ❶
        responses.append(result)
        if len(responses) == len(urls):
            callback(responses)
    for url in urls:
        http_client.fetch(url, callback=_finish_fetch_urls)

def run_experiment(base_url, num_iter=500, callback=None):
    urls = generate_urls(base_url, num_iter)
    callback_passthrou = partial(_finish_run_experiment,
            callback=callback) # ❷
    fetch_urls(urls, callback_passthrou)

def _finish_run_experiment(responses, callback):
    response_sum = sum(len(r.body) for r in responses)
    print response_sum
    callback()

if __name__ == "__main__":
    # ... initialization ...

    _ioloop = ioloop.IOLoop.instance()
    _ioloop.add_callback(run_experiment, base_url, num_iter, _ioloop.stop) #❸

    _ioloop.start()
```

❸ 我们把`_ioloop.stop`作为回调传给`run_experiment`，这样一旦实验完成，它就会为我们关闭 I/O 循环。

❷ 回调类型的异步代码包含了许多偏函数的创建。这是因为我们常常需要保留我们传送过去的原始回调，即使当前我们需要把运行时转移给其他函数。

❶ 有时候玩弄局部域是一种有必要的作恶，那是为了保持状态而又不扰乱全局命名空间。

`gevent` 和 `tornado` 之间另一个有趣的区别是它们内部改变请求调用图的方式。对比图 8-5 和图 8-4，对于 `gevent` 的调用图，我们看到一些区域的对角线

看上去更细，而另一些区域的对角线看上去变得更粗。更细的区域显示出在发起新的请求之前，我们正在等待旧请求结束的那些时间段。更粗的区域代表我们太忙了，以致无法读取来自那些本应该已经结束的请求的响应。这些类型的区域代表了事件循环不能优化工作的时间段：或者对资源利用不足，或者超负荷使用资源。

另外，tornado 的调用图要更均匀得多。这显示出“tornado”能够更好地优化资源使用。这可以归因于许多因素。这里的一个贡献因素就是因为限制并发请求的数量到 100 的信号量机制是内建于 tornado 的，它能够更好地分配资源。还包括以更智能的方式来预分配和复用连接。此外，有许多更小的效果是来自模块就它们与内核的通信方式上的选择，这样是为了协调从多异步操作中接收结果。

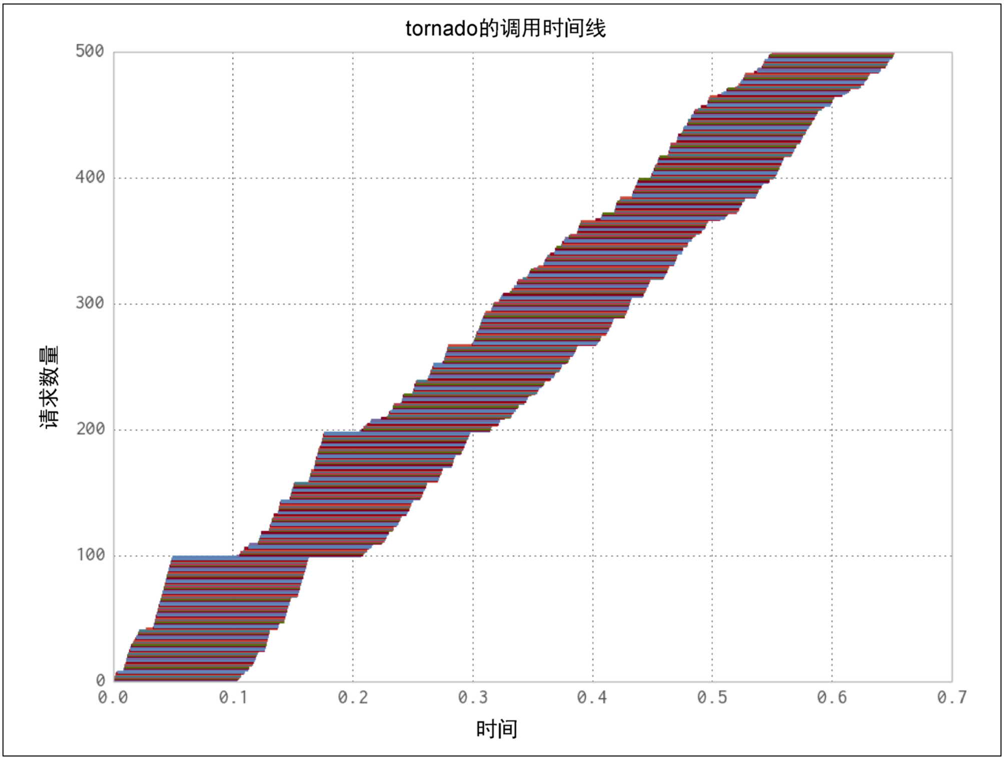


图 8-5　例 8-6 的 HTTP 请求时间表

8.5　AsyncIO

作为对使用异步函数来处理重量级 I/O 系统的风潮的回应，Python 3.4+引入了对老式的 asyncio 标准库模块的改造。这个模块备受 gevent 和 tornado 并发

方法的影响，定义了协程，并可以从协程切换，从而暂停当前函数的执行并允许让其他协程运行。就像在 tornado 中那样，事件循环显式地启动来开始执行协程。此外，Python 3 引入了一个新的关键词——yield from，大大简化了对这些协程的处理（我们不再需要从一个协程中抛出异常来返回值，就如我们在例 8-6 中做的那样）。

值得重视的是 asyncio 库是很低层的，并不对用户提供更高层的功能。例如，尽管有很全面的 socket API，但却没有简单的方法来做 HTTP 请求。作为结果，我们在例 8-8 中选择使用 aiohttp。无论如何，对 asyncio 库的采用正开始增长，辅助模块的前景可能正在飞速变化。

例 8-8　asyncio HTTP 爬虫

```
import asyncio
import aiohttp
import random
import string

def generate_urls(base_url, num_urls):
    for i in range(num_urls):
        yield base_url + "".join(random.sample(string.ascii_lowercase, 10))

def chunked_http_client(num_chunks):
    semaphore = asyncio.Semaphore(num_chunks) # ❶
    @asyncio.coroutine
    def http_get(url): # ❷
        nonlocal semaphore
        with (yield from semaphore):
            response = yield from aiohttp.request('GET', url)
            body = yield from response.content.read()
            yield from response.wait_for_close()
        return body
    return http_get

def run_experiment(base_url, num_iter=500):
    urls = generate_urls(base_url, num_iter)
   http_client = chunked_http_client(100)
   tasks = [http_client(url) for url in urls] # ❸
   responses_sum = 0
   for future in asyncio.as_completed(tasks): # ❹
        data = yield from future
        responses_sum += len(data)
    return responses_sum
```

```
if __name__ == "__main__":
    import time
    delay = 100
    num_iter = 500
    base_url = "http://127.0.0.1:8080/add?name=asyncio&delay={}&".format(delay)
    loop = asyncio.get_event_loop()

    start = time.time()
    result = loop.run_until_complete(run_experiment(base_url, num_iter))
    end = time.time()
    print("{} {}".format(result, end-start))
```

❶ 正如在 gevent 中的例子一样，我们必须使用信号量来限制请求数量。

❷ 我们返回一个新的协程来异步下载文件，并遵从信号量的上锁。

❸ `http_client` 函数返回 futures。为了跟踪进度，我们把 futures 存入一个列表。

❹ 就像用 `gevent` 那样，我们能够等待 futures 准备就绪并去遍历它们。

`asyncio` 模块的一个很大的好处就是与标准库相比熟悉的 API，这样简化了创建辅助模块。我们能够得到与使用 `tornado` 或 `gevent` 相同类型的结果，但是如果我们想要，我们就能更深入地探索软件栈，并能用广泛的支持结构来创建我们自己的异步协议。此外，因为它是一个标准库模块，能够向我们确保这个模块总是遵照 PEP 并且得到了合理的维护[①]。

而且，`asyncio` 库允许我们统一像 `tornado` 和 `gevent` 这样的模块，让它们在相同的事件循环中运行。事实上，Python 3.4 版本的 `tornado` 由 `asyncio` 库作为后台支持。结果就是，尽管 `tornado` 和 `gevent` 有着不同的使用场景，底层的事件循环是统一化的，这就让从一种模式切换到其他中间代码变得轻而易举。你甚至能基于 `asyncio` 模块之上相当容易地创建你自己的包装器，为了以对你正在解决的问题可能是最有效率的方式来与异步操作交互。

尽管它只在 Python 3.4 或更高的版本中[②]得到支持，这个模块至少是一个伟大的标志，将来有更多的工作将放入异步 I/O 中。因为 Python 开始在越来越多的处理流水线中占主导地位（从数据处理到 web 请求处理），这种转变意义深远。

图 8-6 显示了我们 HTTP 爬虫的 `asyncio` 版本的请求时间线。

① Python 增强协议（PEPs）是 Python 社区对于如何变化和如何推进语言的决定。因为它是标准库的一部分，`asyncio` 将总是遵守语言的最新 PEP 标准并且利用任何最新的特性。

② 大多数性能应用程序和模块还是在 Python 2.7 的生态系统中。

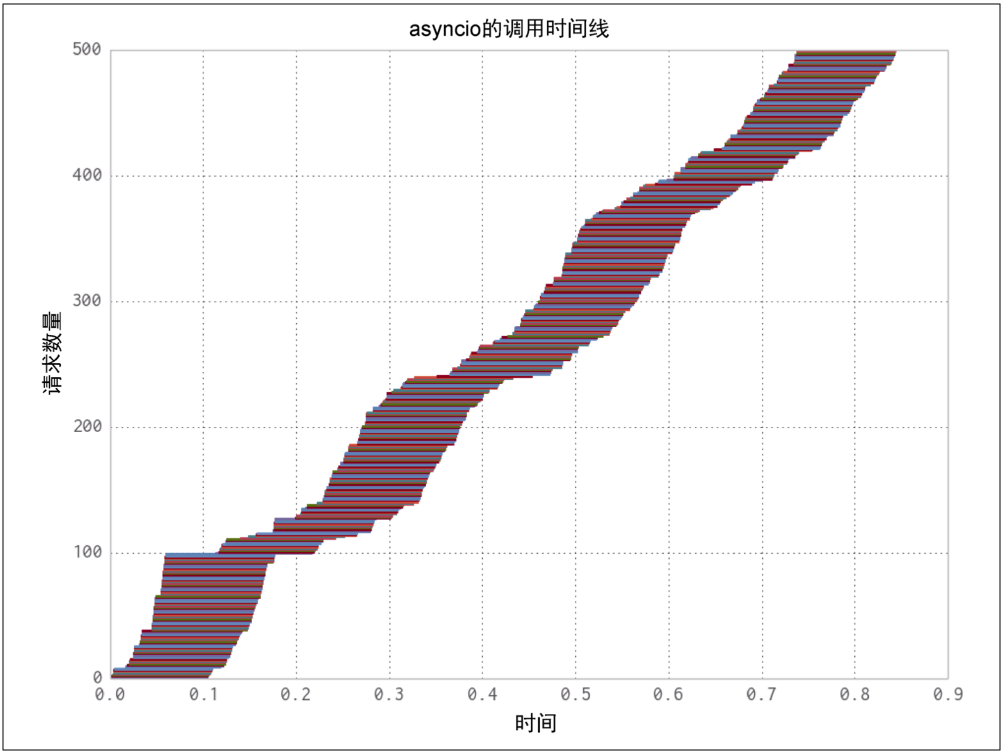


图 8-6　例 8-8 的 HTTP 请求时间表

8.6　数据库的例子

为了让前面的例子更具体，我们还制造了另一个玩具型的问题，主要是 CPU 密集型的但是包含了潜在的限制 I/O 的组件。我们将要计算素数，并把发现的素数存入一个数据库中。数据可以是任意的，问题则具有代表性，代表了你的程序有着要做的任何类型的重量级计算，那些计算的结果必须得存入一个数据库中，偷偷地招来重量级的 I/O 惩罚。我们对数据库施加的限制只有：

- 有 HTTP 的 API，这样我们就能够使用像前面的例子中那样的代码[①]。
- 响应时间在 50 毫秒的级别。
- 数据库能够满足同一时刻处理多个请求[②]。

① 这不是必要的，它只是简化了我们的代码。
② 这对所有的分布式数据库和其他流行的数据库都成立，比如 Postgres、MongoDB、Riak 等。

我们从一些简单的代码开始，计算素数并且每当发现了一个素数，就向数据库的 HTTP API 发起请求：

```
from tornado.httpclient import HTTPClient
import math

httpclient = HTTPClient()
def save_prime_serial(prime):
    url = "http://127.0.0.1:8080/add?prime={}".format(prime)
    response = httpclient.fetch(url)
    finish_save_prime(response, prime)

def finish_save_prime(response, prime):
    if response.code != 200:
        print "Error saving prime: {}".format(prime)

def check_prime(number):
    if number % 2 == 0:
        return False
    for i in xrange(3, int(math.sqrt(number)) + 1, 2):
        if number % i == 0:
            return False
    return True

def calculate_primes_serial(max_number):
    for number in xrange(max_number):
        if check_prime(number):
            save_prime_serial(number)
    return
```

正如我们在串行例子（例 8-3）中的那样，每次数据库存储的请求时间（50 毫秒）没有堆积，并且我们必须为我们所发现的每一个素数付出这个代价。结果就是，搜索到 `max_number = 8192`（产生了 1028 个素数）花费了 55.2 秒。我们知道，无论怎样，正是因为我们的串行请求工作方式，我们至少花费 51.4 秒来做 I/O！所以，只是因为我们正在做 I/O 时暂停了程序，我们浪费了 93%的时间。

作为替代，我们想做的事情就是找到改变我们请求模式的方式，这样我们就能同时异步地发出很多请求，我们就不需要如此难以承担的 I/O 等待。为了做到，我们创建了一个 `AsyncBatcher` 类来为我们处理批量的请求，并在需要时发出请求：

```
import grequests
from itertools import izip

class AsyncBatcher(object):
```

```
    __slots__ = ["batch", "batch_size", "save", "flush"]
    def __init__(self, batch_size):
        self.batch_size = batch_size
        self.batch = []

    def save(self, prime):
        url = "http://127.0.0.1:8080/add?prime={}".format(prime)
        self.batch.append((url,prime))
        if len(self.batch) == self.batch_size:
            self.flush()

    def flush(self):
        responses_futures = (grequests.get(url) for url, _ in self.batch)
        responses = grequests.map(responses_futures)
        for response, (url, prime) in izip(responses, self.batch):
            finish_save_prime(response, prime)
        self.batch = []
```

现在，我们能够以与我们之前所做的同样的方式前进。主要区别仅仅是我们给 `AsyncBatcher` 添加了我们的新素数，并让它来处理什么时候发送请求。此外，既然我们正在批量处理，我们必须确保发送最后那批，即使它还未满（意味着调用 `AsyncBatcher.flush()`）。

```
def calculate_primes_async(max_number):
    batcher = AsyncBatcher(100) # ❶
    for number in xrange(max_number):
        if check_prime(number):
            batcher.save(number)
    batcher.flush()
    return
```

❶ 我们选择以 100 个请求为批次，原因与图 8-3 中所示的那样类似。

随着这个改变，我们能够把计算到 `max_number = 8192` 的运行时间降低到 4.09 秒。这就代表了 13.5 倍的速度提升，而没有做很多的工作。在一个类似实时数据处理的约束环境下，这种额外的速度就可能意味着区分一个系统是能跟上需求还是落后于需求（在这种情况下，需要有一个队列，你会在第 10 章学到这些内容）。

在图 8-7 中，我们能看到这些变化在不同的工作负荷中影响代码的运行时间。异步代码相对串行代码的速度提升是显著的，尽管我们还不是在原始的 CPU 问题上取得的提速。为了完全改进这个问题，我们需要使用像 `multiprocess` 之类的模块来以一个完全独立的进程来处理问题中的 I/O 负担部分，而不会去拖慢问题中的 CPU 运算部分。

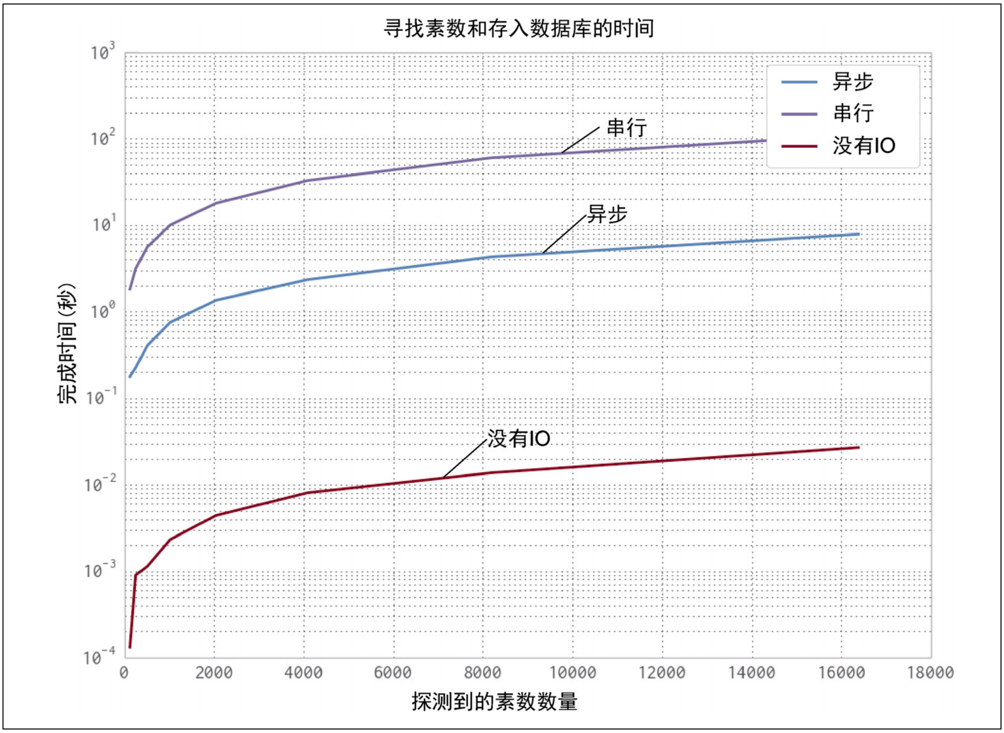


图 8-7　不同数量的素数处理时间

8.7　小结

当解决在现实世界中和生产系统中的问题时，常常需要和一些外部源通信。这个外部源可能是一个在另一台服务器上运行的数据库，另一个工作主机，或者一个提供了必须要处理的原始数据的数据服务。无论哪种情况，你的问题会很快变成 I/O 密集型，意味着对输入/输出的处理占据了大部分运行时间。

并发通过允许你把潜在的多个 I/O 操作交织起来，从而有助于 I/O 密集型的问题。这样就允许你探索 I/O 和 CPU 操作的基本区别来提升整体的运行时间。

就如我们看到的那样，`gevent` 为异步 I/O 提供了最高级别的接口。另一方面，`tornado` 让你手动控制事件循环的运行，允许你使用事件循环来调度你想要的任何类型的任务。最后，在 Python 3.4+中的 `asyncio` 允许完全控制一个异步 I/O 栈。除了各种各样的抽象级别，每个库为它的语法使用了一个不同的范型（差异主要源于在 Python 3 以前缺乏对并发的原生支持以及引入了 `yield from` 声明）。我们推荐从这一系列方法中去获取经验，并基于需要多少低层控制来挑选其中一个。

最后，我们采用过的这 3 个库中有轻微的速度差异。这些速度差异很多都是基于协程的调度方式。例如，`tornado` 做了一件极好的工作来快速启动异步操作并快速让协程继续运行。另一方面，尽管 `asyncio` 看上去运行得稍微糟了一点，但是它允许访问更低层的 API，并能够动态调整。

在下一章中，我们会采用这个来自于 I/O 密集型问题的交织计算的概念，并把它应用于 CPU 密集型问题。使用了这个新的力量，我们将不但能够同时运行多个 I/O 操作，而且也能够同时运行许多计算型的问题。这种能力将允许你开始创建完全可扩展的程序，在其中，我们可以通过仅仅增加更多的能够分别处理分块问题的计算资源来获得更多的速度提升。

第 9 章

multiprocessing 模块

读完本章之后你将能够回答下列问题

- `multiprocessing` 模块提供了什么？
- 进程和线程的区别是什么？
- 我该如何选择合适大小的进程池？
- 我该如何使用非持久队列来处理工作？
- 进程间通信的代价和好处是什么？
- 我该如何用多 CPU 来处理 numpy 数据？
- 为什么我需要加锁来避免数据丢失

CPython 默认没有使用多 CPU。一部分原因是 Python 是被设计用于单核领域，另一部分原因是实际上有效的并行化是相当困难的。Python 给我们提供了工具，但是让我们自己来做出选择。然而，看到你的多核机器上只使用了一个 CPU 来长期运行一个进程是痛苦的，所以在本章中我们会立即检视各种方法来使用所有的机器核。

备忘

值得注意的是我们在上面提到的是 CPython（我们所有人都使用的通用实现）。在 Python 语言中，没有什么东西阻止使用多核系统。CPython 的实现不能有效使用多核，但是其他实现（例如，具有即将到来的软件事务内存的 PyPy）可能不会被这个约束所束缚。

我们生活于一个多核世界中——笔记本电脑上普遍为 4 核，桌面电脑上的 8 核的配置将很快流行起来，并且 10-、12-和 15 核 CPU 的服务器也存在。如果你的工作能被拆分成运行于多核 CPU，而又不花费太多工程方面的努力，那么这是一个要考虑的明智的方向。

当习惯于在一个 CPU 集上并行化问题时，你就能期待用 *n* 核达到 *n* 倍（nx）的速度提升。如果你有一个 4 核的机器，并且能为你的任务使用全部的 4 核，它就有可能以原来运行时间的四分之一来跑完。你不可能看到一个大于 4 倍的提速。在实践中，你可能会看到 3 到 4 倍的增益。

每一个额外的处理将会增加通信的开销和减少可使用的内存，所以你很少会得到一个完全的 *n* 倍的提速。取决于你正在解决的问题，通信开销甚至可以变得很大以致于你能够看到很明显的减速。这些类型的问题常常是存在于任何类型的并行编程中的复杂性，并且通常需要去改变算法。这就是并行编程常常被认为是一种艺术的原因。

如果你不熟悉 Amdahl 定律，那就值得去阅读一些背景材料。这个定律揭示了如果你的代码只有一小部分能够并行化，那就和你给它用多少 CPU 无关。整体上，它还是无法运行得更快。在你得到回报减弱的要点之前，即使你的程序在运行时有很大一部分能够并行化，也只有有限数量的 CPU 能被有效利用来使整体进程运行得更快。

`multiprocessing` 模块让你使用基于进程和基于线程的并行处理，在队列上共享任务，以及在进程间共享数据。它主要是集中于单机多核的并行（对多机并行来说，有更好的选择）。一个很普遍的用法就是针对 CPU 密集型的问题，在一个进程集上并行化一个任务。你可能也用它来并行化 I/O 密集型问题，但是就如我们在第 8 章所见的那样，有更好的工具来处理这类问题（例如，在 Python 3.4+中的新 `asyncio` 模块和在 Python 2+中的 `gevent` 或者 `tornado`）。

备忘

OpenMP 是一个低层的多核接口——你可能想知道是集中精力于它上面还是于 `multiprocessing` 上面。我们在第 7 章中与 Cython 和 Pythran 在一起介绍过它，但是我们在第 7 章中并没有全面涉猎它。`multiprocessing` 在一个更高的层次上工作，共享 Python 的数据结构，而 OpenMP 一旦被编译成 C 后，就使用 C 的原生对象（例如，整型数和浮点数）来工作。它只有在你编译你的代码时才有意义去使用。如果你不去编译（例如，如果你正使用高效的 `numpy` 代码并想要在多核上运行），那么坚持使用 `multiprocessing` 可能是正确的途径。

为了并行化你的任务，你必须要以比编写一个串行程序的普通方式稍微有别一点的方式去思考。你也必须接受更大的困难去调试一个并行任务——它常常是很令人沮丧的。我们要推荐尽可能地让并行保持简单（即使你压榨不出你的机器的每一滴最后的力量），这样你就会保持高速的开发。

一个特别困难的主题就是在并行系统中共享状态——凭感觉这好似应该简单，但是却带来了很多开销，并且难以做正确。有许多应用案例，每一个都有不同的妥协，所以肯定没有针对所有情况的解决方案。在 9.5 节，我们将会用一只眼盯着同步的开销来遍历下状态共享。避免共享状态会让你的生活变得简单很多。

事实上，一个算法能够几乎全凭有多少状态必须要共享来分析出它在并行环境中表现如何。例如，如果我们有多个 Python 进程全都是解决一样的问题，而没有彼此间相互通信（一种已知为窘迫并行的情况），那当我们增加越来越多的 Python 进程时，就不会招致多大的惩罚。

另一方面，如果每一个进程需要和所有其他 Python 进程来通信，那么通信开销将会慢慢让处理变得不堪重负，拖慢了事情。这意味着当我们增加越来越多的 Python 进程时，我们实际上减慢了整体性能。

作为结果，有时必须要做一些反直觉的算法改动来有效解决并行问题。例如，当解决并行扩散方程（第 6 章）时，每一个进程实际上做了另一个进程也在做的冗余工作。这种冗余降低了所需的通信量，并且提高了整体的计算速度！

multiprocessing 模块有一些典型的工作：

- 用进程或池对象来并行化一个 CPU 密集型任务。
- 用哑元模块（奇怪的称呼）在线程池中并行化一个 I/O 密集型任务。
- 由队列来共享捎带的工作。
- 在并行工作者之间共享状态，包括字节、原生数据类型、字典和列表。

如果你从一种使用线程来做 CPU 密集型任务（例如，C++或 Java）的语言中转过来，那么你应该知道尽管在 Python 中的线程是 OS 原生的（它们不是模拟出来的，它们是真实的操作系统线程）,它们被全局解释锁（GIL）所束缚，所以同一个时刻只有一个线程可以和 Python 对象交互。

通过使用进程，我们并行运行了一定数量的 Python 解释器，每一个进程都有私有的内存空间，有自己的 GIL，并且每一个都串行运行（所以没有 GIL 之间的竞争）。

这是在 Python 中提升 CPU 密集型任务速度的最简单的方式。如果我们需要共享状态，那么我们就需要增加一些通信开销。我们在 9.5 节中会进行探索。

如果你用 numpy 数组工作，你可能想知道你是否可以创建一个更大的数组（例如，一个大 2 维矩阵），以及是否可以让进程并行工作于分段数组。你可以，但是通过试错难以发现怎样做，所以在 9.6 节中，我们会经历一遍在 4 个 CPU 之间共享一个 6.4GB 的 `numpy` 数组。与其传送部分拷贝数据（至少会让在 RAM 中的工作集大小翻倍，并且会产生巨大的通信开销），我们在进程间共享底层的数组字节。这是一个在一台机器上的本地工作者之间共享大数组的理想方式。

备忘

这里，我们是在基于*nix 的机器上讨论 `multiprocessing`（本章是用 ubuntu 来写的，代码应该能不做改动在 Mac 上运行）。对于 Windows 机器，你应该检查官方文档。

在本章接下来中，我们会硬编码一定数量的进程（`NUM_PROCESSES=4`）来在 Ian 笔记本上匹配 4 个物理核。默认情况下，`multiprocessing` 将使用它能见到的尽可能多的核（机器有 8 核——4 CPU 和 4 超线程）。通常你会避免硬编码进程的数量来创建，除非你有特别的要求来管理你的资源。

9.1 multiprocessing 模块综述

multiprocessing 模块在 Python 2.6 中被引入，通过采用已经存在的 `pyProcessing` 模块，把它合入 Python 的内置库集合中。它的主要组件是：

进程

　　一个当前进程的派生（forked）拷贝，创建了一个新的进程标识符，并且任务在操作系统中以一个独立的子进程运行。你可以启动并查询进程的状态并给它提供一个目标方法来运行。

池

　　包装了进程或线程。在一个方便的工作者线程池中共享一块工作并返回聚合的结果。

队列

　　一个先进先出（FIFP）的队列允许多个生产者和消费者。

管理者

一个单向或双向的在两个进程间的通信渠道。

ctypes

允许在进程派生（forked）后，在父子进程间共享原生数据类型（例如，整型数、浮点数和字节数）。

同步原语

锁和信号量在进程间同步控制流。

备忘

在 Python 3.2 中引入了 `concurrent.futures` 模块（通过 PEP 3148），它通过一个更简单的基于 Java 的 `java.util.concurrent` 接口提供了 `multiprocessing` 的核心行为。它向后兼容于更早的 Python 版本。我们在这里不去提它，因为它不像 `multiprocessing` 那样灵活，但是我们怀疑随着 Python 3+越来越多地被采用，我们将看到它有朝一日会取代 `multiprocessing`。

在本章余下部分，我们会介绍一组例子来演示使用这个模块的普遍方法。

我们将使用蒙特卡罗方法由一个进程池或者线程池来估算 pi 值，使用常规的 Python 和 numpy。这是一个简单问题，具有很好理解的复杂性，所以它能轻松地并行化。我们也能够从使用 numpy 的线程中看到一个预料之外的结果。接下来，我们会使用相同的池方法来搜索素数。我们会调查搜索素数的不可预测的复杂性，并且看看我们怎样能有效（和无效!）地拆分工作量来最大化地利用我们的计算资源。我们会通过转移到队列来完成素数搜索，在那里我们会引入 `Process` 对象来取代池并且使用一个工作和毒药列表来控制工作者的生命周期。

接下来，我们会通过处理进程间通信（IPC）来验证一个小的可能的素数集合。通过在多个 CPU 之间拆分每一个数字的工作负载，如果找到了一个因子，我们就使用 IPC 来提早结束搜索，这样我们能够显著地击败单个 CPU 搜索进程的速度。我们将会涉及共享 Python 对象、OS 原语和一个 Redis 服务器来调查每一种方法在复杂性和扩展性上面的妥协。

我们可以在 4 个 CPU 之间共享一个 6.4GB 的 `numpy` 数组，从而拆分一个巨大的工作负载而不用拷贝数据。如果你有可并行化操作的大数组，那么这个技术应该为你带来巨大的速度提升，因为你可以在 RAM 中分配更少的空间，拷贝更少的数据。

最后，我们会看看在进程间同步存取一个文件和一个变量（作为一个 Value）而不破坏数据，从而演示如何正确地锁住共享状态。

备忘

PyPy（在第 7 章讨论过）全面支持 `multiprocessing` 库，接下来的 CPython 例子（尽管在写作时没有 `numpy` 的例子）使用 PyPy 全都运行得快很多。如果你只使用 CPython 代码（没有 C 扩展或者更多的复杂库）来做并行处理，那么 PyPy 可能就会对你快速取胜。

本章（和整本书）集中于 Linux 上。Linux 具有派生（`fork`）进程，通过克隆父进程来创建新进程。Windows 缺少 `fork`，所以 `multiprocessing` 模块施加了一些 Windows 特有的约束，如果你要使用 Windows 平台，我们劝你要检查一下。

9.2 使用蒙特卡罗方法来估算 pi

我们可以通过向一个由一个单位圆所代表的飞镖靶子投掷几千枚虚构的飞镖来估算 pi。落入圆圈的边缘内和边缘外的飞镖数量之间的关系将允许我们来趋近 pi。

这是第一个理想问题，因为我们可以把整个工作负载在一定数量的进程间均匀地拆分，每一个运行在一个独立的 CPU 上面。每一个进程将会同时结束，因为每一个进程的工作负载是均等的，所以在我们给问题增加新的 CPU 和超线程时，我们可以调查可得到的速度提升。

在图 9-1 中，我们朝单位圆投掷了 10 000 枚飞镖，其中一定比例的飞镖落入了画出来的单位圆的四等分之内。这个估算很坏——10 000 枚飞镖投掷没有可靠地给出我们三小数位的结果。如果你运行自己的代码，会看到每一轮估算值在 3.0 到 3.2 之间变化。

为了对第一个三小数位有信心，我们需要产生 10 000 000 个随机飞镖投掷。这是低效的（存在更好的 pi 值估算方法），但是它相当方便地演示了使用 `multiprocessing` 做并行化的好处。

随着蒙特卡罗方法，我们使用 Pythagorean 理论来测试一个飞镖是否在我们的圆圈内着落：

$$\sqrt{(x^2 + y^2)} \leqslant 1^2$$

因为我们使用了一个单位圆，所以我们能够通过移除平方根操作（1^2=1）来优化，给我们留以一个简化的表达式去实现：

$$x^2 + y^2 \leqslant 1^2$$

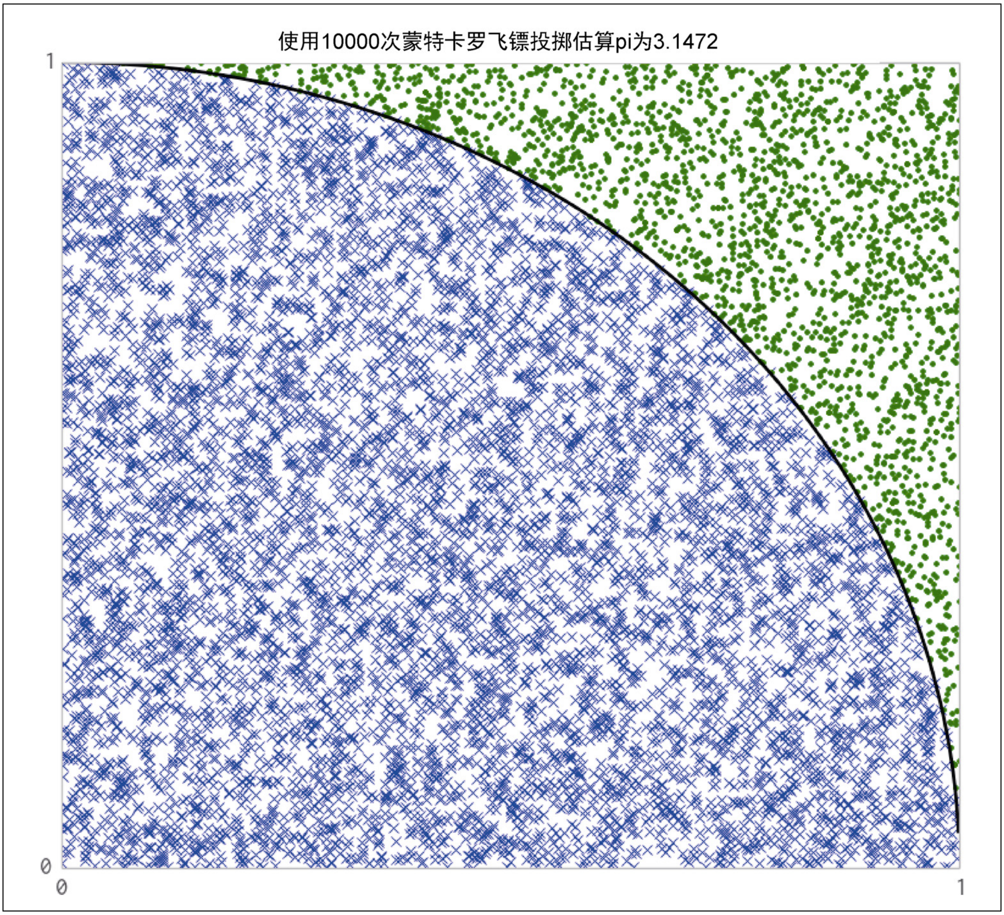


图 9-1　使用蒙特卡罗方法来估算 pi

我们在例 9-1 中会看到一个循环的版本。我们会实现一个普通的 Python 版本，以及以后实现一个 numpy 版本，我们都将使用线程和进程来并行化问题。

9.3　使用多进程和多线程来估算 pi

理解一个普通的 Python 实现更简单，所以我们在本节中会以它开始，在一个循环

中使用浮点对象。我们会使用多进程来用到所有可利用的 CPU 并行化它，并且当我们使用更多的 CPU 时，我们会把机器的状态可视化。

9.3.1 使用 Python 对象

Python 的实现容易模仿，但是它带来了一个开销，因为每个 Python 浮点对象必须要依次被管理、引用和同步化。这个开销减慢了我们的运行时间，但是它带给了我们思考时间，因为很快就能实现好。通过并行化这个版本，我们几乎没有什么额外工作却得到了附加的速度提升。

例 9-2 显示了 Python 例子的三个实现：

- 不使用 `multiprocessing`（称之为“串行”）。
- 使用多线程。
- 使用多进程。

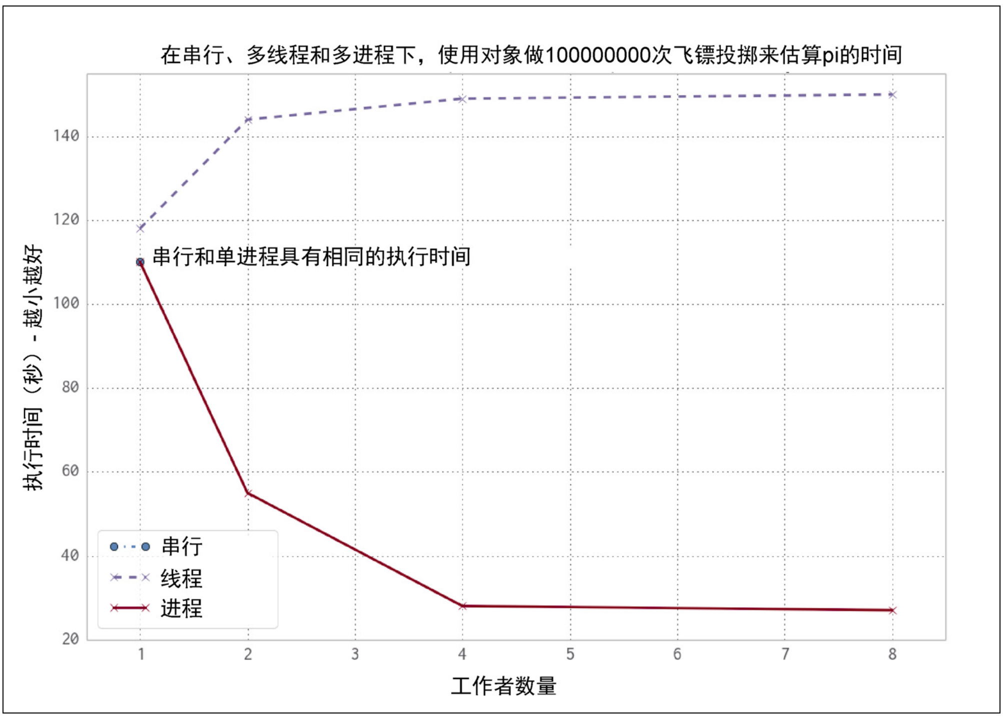


图 9-2 工作于串行、多线程和多进程中

当我们使用超过一个线程或进程时，我们让 Python 计算相同总数量的飞镖投掷，让 Python 把工作均匀地在工作者之间划分。如果我们使用 Python 实现，想要总共

投掷 100 000 000 次飞镖，并且我们使用两个工作者，那么我们就会让这两个线程或进程都给每个工作者生成 50 000 000 次飞镖投掷。

使用一个线程大概花费 120 秒。使用两个或更多线程花费更久。通过使用两个或更多进程，我们缩短了运行时间。不使用多进程或多线程（串行实现）和用单进程运行的效果一样。

通过使用多进程，当在使用双核或四核的 Ian 笔记本电脑上运行时，我们得到了一个线性的加速。对于 8 个工作者的情况，我们会使用英特尔的超线程技术——笔记本电脑只有 4 个物理核，所以我们几乎不能通过运行 8 个进程来获得额外的速度提升。

例 9-1 展示了我们 Python 版本的 pi 估算器。如果我们使用多线程，每条指令被 GIL 所束缚，所以尽管每个线程能够在独立的 CPU 上运行，但是它只会在没有其他线程运行时才会运行。进程的版本不受这种约束的束缚，因为每个派生（fork）出来的进程都有一个私有的 Python 解释器运行在一个单独线程中——因为没有共享对象，所以没有 GIL 竞争。我们使用了 Python 内置的随机数生成器，但是看看 9.3.2 节中的一些注意点，是关于并行化随机数序列存在的风险。

例 9-1　在 Python 中使用一个循环来估算 pi

```
def estimate_nbr_points_in_quarter_circle(nbr_estimates):
    nbr_trials_in_quarter_unit_circle = 0
    for step in xrange(int(nbr_estimates)):
        x = random.uniform(0, 1)
        y = random.uniform(0, 1)
        is_in_unit_circle = x * x + y * y <= 1.0
        nbr_trials_in_quarter_unit_circle += is_in_unit_circle

return nbr_trials_in_quarter_unit_circle
```

例 9-2 展示了`__main__`代码块。注意我们在启动定时器前就构建了池。生成线程相对快速，生成进程则涉及一个派生拷贝（fork），这要花费可度量的不到一秒的时间。我们在图 9-2 中忽略了这个开销，因为它占整体执行时间的一个微不足道的部分。

例 9-2　使用一个循环来估算 pi 的 main

```
from multiprocessing import Pool
...

if __name__ == "__main__":
    nbr_samples_in_total = 1e8
```

```
nbr_parallel_blocks = 4
pool = Pool(processes=nbr_parallel_blocks)
nbr_samples_per_worker = nbr_samples_in_total / nbr_parallel_blocks
print "Making {} samples per worker".format(nbr_samples_per_worker)
nbr_trials_per_process = [nbr_samples_per_worker] * nbr_parallel_blocks
t1 = time.time()
nbr_in_unit_circles = pool.map(calculate_pi, nbr_trials_per_process)
pi_estimate = sum(nbr_in_unit_circles) * 4 / nbr_samples_in_total
print "Estimated pi", pi_estimate
print "Delta:", time.time() - t1
```

我们创建了一个包含 `nbr_estimates` 的列表，被工作者的数量整除。这个新参数将会被送给每一个工作者。在执行之后，我们会收到相同数量的返回结果，我们会把这些结果累加起来去估算单位圆内的飞镖数量。

我们从 `multiprocessing` 导入了进程池。我们本也能用 `from multiprocessing.dummy import Pool` 来得到一个线程的版本—— “dummy”这个名字相当误导人（我们坦承并不理解为什么用这种方式来命名），它仅仅是一层薄薄的 threading 模块的包装器来表示与进程池相同的接口。

警告

值得注意的是我们创建的每一个进程从系统消耗了一些 RAM。我们可以期望一个使用标准库的派生（fork）进程占用 10MB 到 20MB 数量级的 RAM；如果你使用了很多库和数据，那么你可以期望每一个派生（fork）拷贝进程会占用几百兆字节。在一个有 RAM 约束的系统中，这可能是一个明显问题——如果你用完了 RAM，系统就转而使用硬盘的交换空间，那么任何并行优势将会在慢速的 RAM 和硬盘之间的来回换页中发生巨大的损失。

下面的图标绘了 Ian 的笔记本电脑的四个物理核平均 CPU 利用率以及它们相关的四个超线程（每个超线程运行于一个物理核上的未利用的硅片上）。这些图标所采集的数据包括了第一个 Python 进程的启动时间以及启动多个子进程的开销。CPU 采集器记录了笔记本电脑的全部状态，而不仅仅是被这个任务所使用的 CPU 时间。

注意下面的图表使用了一个不同的计时方法来创建，比图 9-2 中的采样率更低，所以整体运行时间稍稍长了一点。

图 9-3 中采用只有一个进程的进程池（和父进程一起运行）的执行表现显示出当创建进程池时一开始有几秒的开销，接着一个稳定的接近于 100%的 CPU 利用率贯穿于整个运行过程中。我们有效地用一个进程来使用了一核。

接着我们会增加第二个进程，实际说来就是 `Pool(processes = 2)`。就如你能在图 9-4 中看到的那样，增加第二个进程大概把执行时间缩短到一半，即 56 秒，并且两个 CPU 被充分地占用。这是我们能期待的最好结果——我们已经有效地使用了所有的新计算资源并且我们没有让其他开销来减慢任何运行速度，比如通信、磁盘换页，或者想要使用相同 CPU 的竞态进程间的竞争。

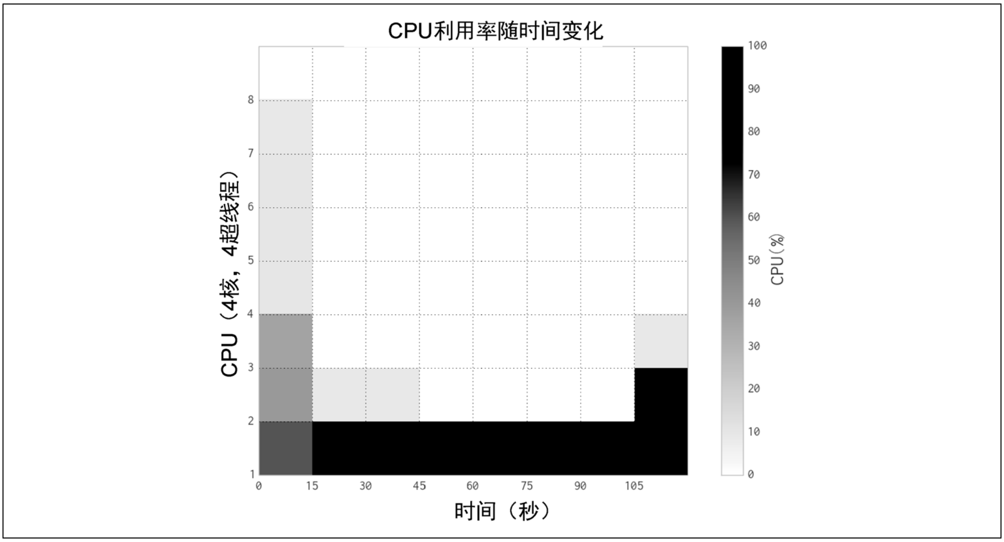


图 9-3　使用 Python 对象和一个进程来估算 pi

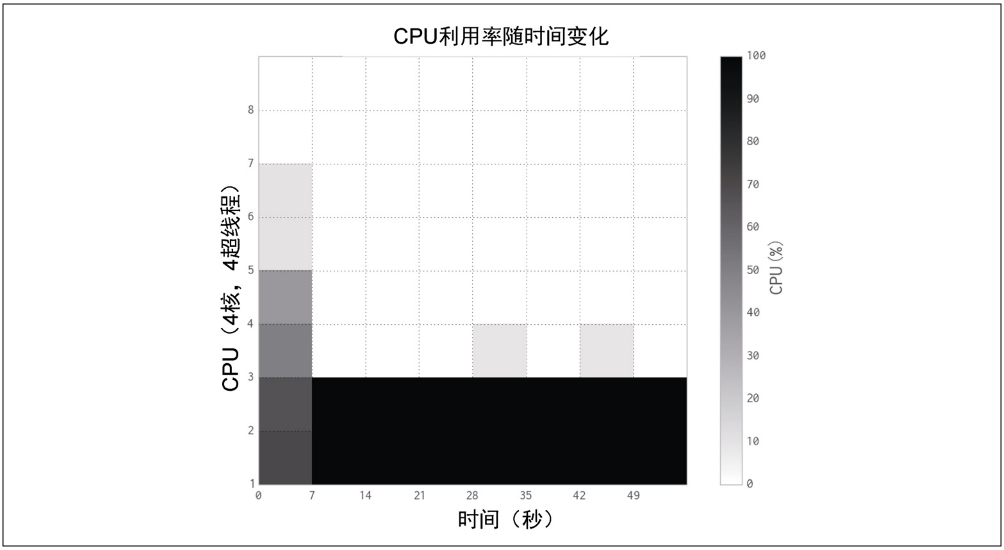


图 9-4　使用 Python 对象和两个进程来估算 pi

图 9-5 显示了当使用全部 4 个物理核后的结果——现在我们正在使用这台笔记本电脑的全部能量。执行时间大约是单进程版本的四分之一，在 27 秒左右。

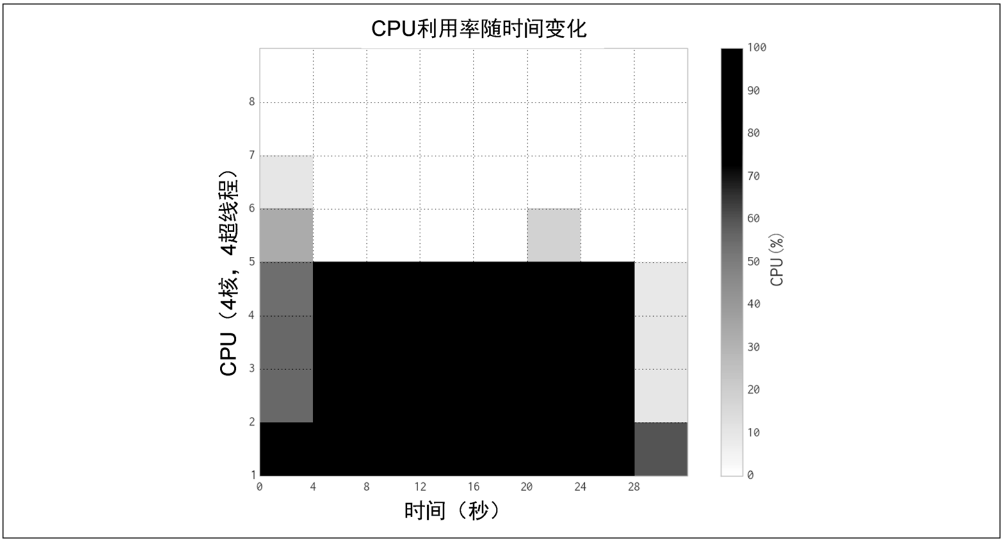


图 9-5　使用 Python 对象和 4 个进程来估算 pi

通过切换到 8 个进程，就如在图 9-6 中所见的那样，我们无法再达到远超 4 个进程版本的速度提升。那是因为 4 个超线程只能够从 CPU 上的闲置硅片中榨出一点点额外的处理能力，4 个 CPU 已经是最大化地得到利用了。

这些图表显示出我们在每一步正有效地利用了超出可用 CPU 的资源，超线程资源是一个很小的附加值。使用超线程的最大问题就是 CPython 使用了很多 RAM——超线程不是缓存友好的，所以对每个芯片上的剩余资源的利用效率很低。就如我们在下一节所见的那样，numpy 更好地利用了这些资源。

备忘

在我们的经验中，如果有足够的剩余计算资源，超线程能够给出直达 30%的性能收益。例如，如果你有一个混合了浮点数和整数的算术运算，而不仅仅是我们这里的浮点数运算，那么它就起效果了。通过混合资源需求，超线程能够调度更多的 CPU 硅片来并发工作。一般情况下，我们把超线程视作一个附加值，而不是一个优化目标资源，因为增加更多的 CPU 或许比调整你的代码（增加了支持开销）来得更经济。

现在我们会切换到使用一个进程中的多线程，而不是多进程。就如你会看到的那样，由“GIL 竞争”所导致的开销实际上让我们的代码运行得更慢了。

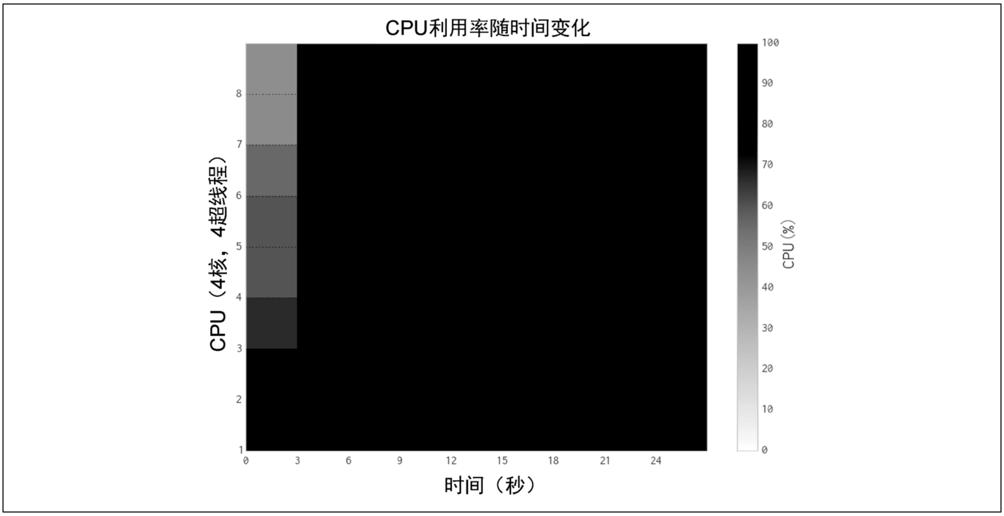


图 9-6　使用 Python 对象和 8 个进程来估算 pi 仅有微不足道的额外收益

图 9-7 显示出两个线程在一个使用 Python 2.6 的双核系统上竞争（会产生和 Python 2.7 相同的效果）——这是从 David Beazley 的博客文章中得到允许所采用的 GIL 的竞争图，“Python GIL 的可视化”。更深的红色调（彩图见本书配套网站，下同）显示了 Python 线程在不断重复地企图获得 GIL 但是失败了。更浅的绿色调代表一个运行中的线程。白色显示了一个线程闲置（idle）的大略周期。我们可以看到当给 CPython 中的 CPU 密集型任务添加多线程时，会有一个开销。David Beazley 在“理解 Python GIL”中做出了解释。Python 中的多线程对于 I/O 密集型任务有优势，但是对 CPU 密集型问题则是一个糟糕的选择。

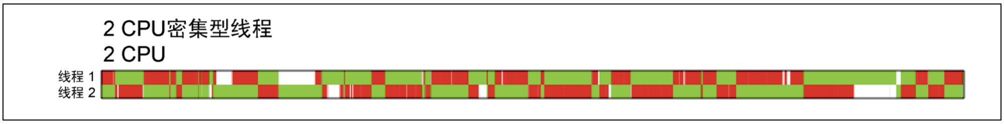


图 9-7　Python 线程在一个双核机器上竞争

每当一个线程被唤醒并设法获取 GIL（无论是否可用）时，它就使用了系统资源。如果一个线程忙碌，那么其他线程将重复不断地唤醒并设法获取 GIL。这些重复的企图变得代价昂贵。David Beazley 有一个相互交互的标绘图集来演示这个问题，你可以放大来看看在多个 CPU 的多线程上每一个失败的获取 GIL 的企图。注意这只是多线程运行于一个多核系统上的问题——一个具有多线程的单核 CPU 没有“GIL 竞争”。这在 David 网站上的 4 线程可缩放视觉图中很容易见到。

如果线程没有在竞争 GIL，但是却有效率地把 GIL 传来传去，那么我们就不要期望

看见任何的深红色调。取而代之的是，我们可能会期待正处于等待中的线程继续等待而又不消耗资源。避免了争用 GIL 就会使整体运行时间缩短，但是因为 GIL，它还是不比使用一个单线程来得更快。如果没有 GIL，每个线程就会并行运行而没有任何的等待，这样线程就会充分利用系统的所有资源。

值得注意的是CPU密集型问题上多线程的负面效果在Python 3.2+中得到了合理的结果：

> 串行化地去执行并发运行的 Python 线程的机制（就是通常所知的 GIL 或者全局解释锁）已经被重写了。在这些目标中，有更可预测的切换时间间隔和降低了因为锁竞争及随之带来的系统调用的数量而引起的开销。“检查时间间隔”的概念允许放弃线程切换并由一个用秒级来表达的绝对时长来取代。
>
> ——Raymon Hettinger

图 9-8 展示了与我们在图 9-5 中所用的相同的代码运行结果，但是用线程来取代了进程。尽管正在使用一定数量的 CPU，但每一个 CPU 却只是稍稍分享了工作负载。

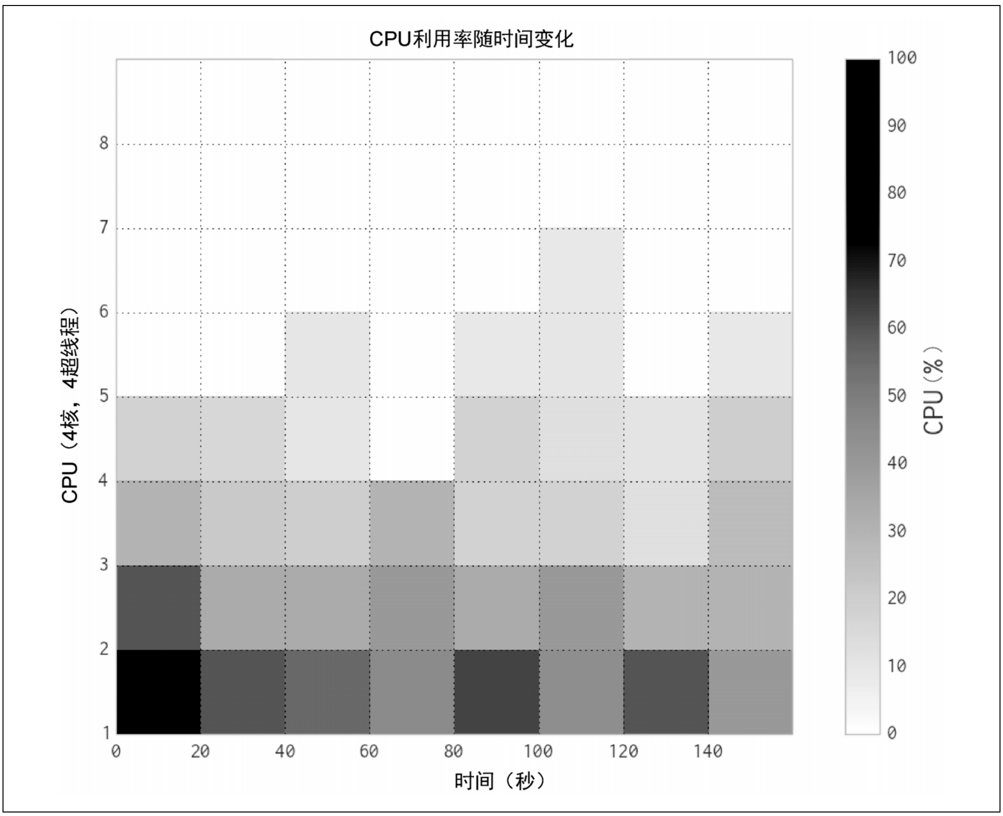


图 9-8 使用 Python 对象和 4 个线程来估算 pi

如果每个线程正在没有 GIL 的情况下运行，那么我们会看到在 4 个 CPU 上 100% 的 CPU 利用率。但有了 GIL，每个 CPU 就只是被部分使用（归因于 GIL），此外由于 GIL 竞争，它们运行得比我们所期望的还要慢。

对比图 9-3，每个进程执行相同的工作用了差不多 120 秒而不是 160 秒。

9.3.2　并行系统中的随机数

生成一个良好的随机数序列是一个困难的问题，如果你设法自己做，就很容易出错。并行地快速得到一个良好的序列甚至更难——突然间你就不得不担心你是否在并行进程中取到了重复性或者相关性的序列。

我们在例 9-1 中已经使用了 Python 内置的随机数生成器，在下一节的例 9-3 中我们会使用 `numpy` 的随机数生成器。在这两种情况下，随机数生成器在它们的派生拷贝（fork）进程中做种子。对于 Python 的 `random` 例子而言，做种子由 `multiprocessing` 内部来处理——如果在派生拷贝（fork）期间，它看到 `random` 在名字空间中，那么它就在每一个新进程中强制调用来为随机数生成器做种子。

在接下来的 `numpy` 例子中，我们必须要显式地去做。如果你忘了给使用 `numpy` 的随机数序列做种子，那么你派生拷贝（fork）的每一个进程将生成一个完全一模一样的随机数序列。

如果你关心并行进程所使用的随机数的质量，那我们就劝你去研究这个主题，因为我们不会在这里讨论。也许 `numpy` 和 Python 的随机数生成器已经足够好，但是如果显著的成果要取决于随机序列的质量（例如，对医疗或金融系统来说），那么你必须要在这个领域攻读。

9.3.3　使用 numpy

在本节中，我们会切换到使用 `numpy`。我们的飞镖投掷问题是理想的 `numpy` 矢量化操作——我们生成了相同的估算值超过 50 次，比之前的 Python 例子要快。

当解决相同的问题时，`numpy` 比纯 Python 要快的主要原因是它在 RAM 的连续块中以一个很低级的层次来创建和操控相同的对象类型，而不是创建许多更高层次的 Python 对象，每一个高级对象需要独立的管理和寻址。

因为 `numpy` 对缓存更友好，所以我们在使用 4 个超线程时，也会得到一个小小的速度提升。我们用纯 Python 版本无法得到这种速度提升，因为更大的 Python 对象没有有效地使用缓存。

在图 9-9 中，我们看到了三个场景：

- 不用 multiprocessing（称之为“串行”）。
- 使用多线程。
- 使用多进程。

串行和单工作者版本以相同的速度运行——没有用 numpy 来使用多线程的开销（当然只有一个工作者，也就没有速度提升）。

当使用多进程时，我们看到每一个附加的 CPU 上一个经典的 100%的利用率。结果就是在图 9-3、图 9-4、图 9-5 和图 9-6 中所展示的标绘图的镜像，但是使用 numpy，代码运行速度明显快了很多。

有趣的是，对线程版本来说，使用更多的线程就运行得更快——这与纯 Python 的情况表现相反，纯 Python 版本的多线程反而让示例运行得更慢。就如在 SciPy wiki 上所讨论的那样，通过工作于 GIL 之外，numpy 能够用多线程达到相同级别的额外加速。

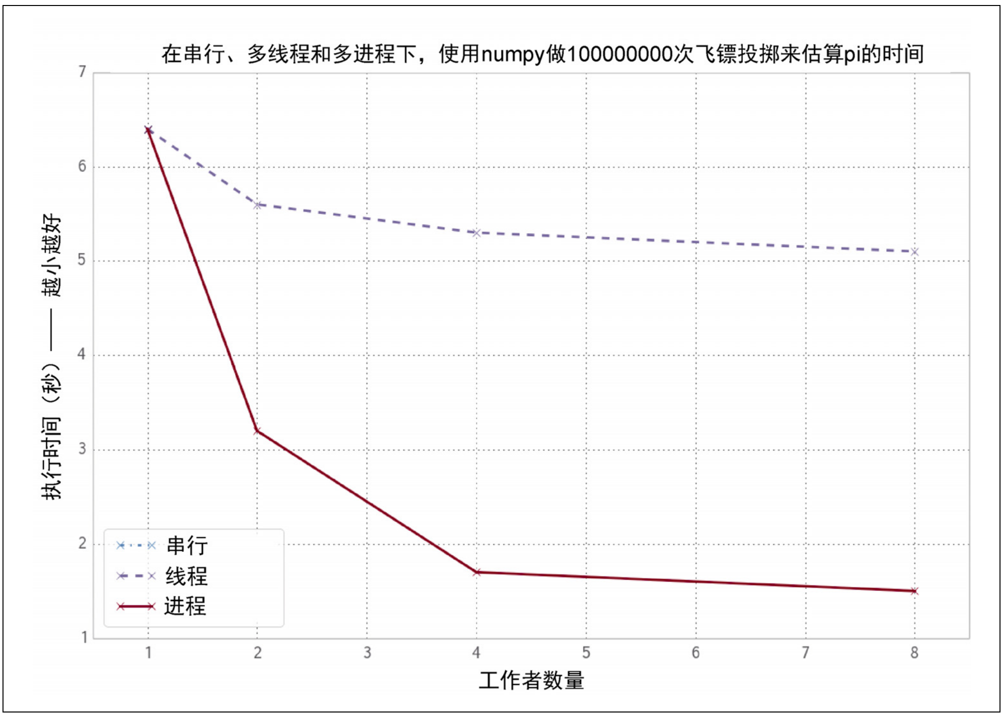


图 9-9　使用 numpy 工作于串行、多线程和多进程

使用多进程给我们一个可预测的速度提升，就如在纯 Python 的例子中所做的那样。第二个 CPU 让速度翻倍，而使用 4 个 CPU 则让速度提高 4 倍。

例 9-3 展示了我们代码的矢量化形式。注意随机数生成器在这个函数被调用的时候做种子。对于线程的版本来说，这不是必须的，因为每个线程共享了相同的随机数生成器，并且它们串行地去访问该生成器。对于进程版本来说，因为每个新进程都是一个派生进程（fork），所以所有派生（folk）的版本将会共享相同的状态。这意味着在每个随机数调用中会返回相同的序列！调用 `seed()`应该确保每个派生（folk）进程生成一个唯一的随机数序列。回头看看 9.3.2 节，有一些关于并行化随机数序列的风险的提示。

例 9-3　使用 numpy 估算 pi

```
def estimate_nbr_points_in_quarter_circle(nbr_samples):
    # set random seed for numpy in each new process
    # else the fork will mean they all share the same state
    np.random.seed()
    xs = np.random.uniform(0, 1, nbr_samples)
    ys = np.random.uniform(0, 1, nbr_samples)
    estimate_inside_quarter_unit_circle = (xs * xs + ys * ys) <= 1
    nbr_trials_in_quarter_unit_circle = np.sum(estimate_inside_quarter_unit_circle)
    return nbr_trials_in_quarter_unit_circle
```

一个简短的代码分析显示当用多线程运行并且很好地并行化调用`(xs * xs + ys * ys) <= 1`时，在这台机器上对 random 的调用还是运行得慢了一点。调用随机数生成器是受制于 GIL 的，因为内部的状态变量是一个 Python 对象。

理解这个过程是基础而可靠的：

1．注释掉所有的 `numpy` 行，并且用没有线程的串行版本来运行。运行几次并在`__main__`中使用 `time.time()`来记录执行时间。

2．添加回去一行（首先，我们增加 `xs = np.random.uniform(…)`，接着运行几次，再次记录完成时间）。

3．添加回去下一行（现在，增加 `ys = …`），再次运行并记录完成时间。

4．重复地添加回去，包含 `nbr_trials_in_quarter_unit_circle = np.sum(…)`行。

5．再次重复这个过程，但这次有 4 个线程。逐行重复。

6．比较无线程时和有 4 个线程时每一个步骤的运行时间差别。

因为我们并行运行代码，所以就更难使用像 line_profiler 或者 cprofile 之类的工具了。记录原始运行时间并观察不同配置的表现差异要花一点耐心，但是却给出了确凿的证据来引出结论。

> **备忘**
>
> 如果你想要理解串行调用 uniform 的行为，那就看一看在 numpy 源码中的 mtrand 代码并在 mtrand.pyx 中跟踪调用 uniform。如果你以前不曾看过 numpy 的源码，这就是一个有用的练习。

构建 numpy 时所用的库对有些并行化的机会是重要的。取决于构建 numpy 时所用的基础库（例如，Intel 的 Math Kernel Library 或者 OpenBLAS 是否包含在内了），你会看到不同的加速行为。

你可以使用 numpy.show_config()来检查 numpy 的配置。如果你对可能性好奇，在 StackOverflow 上有一些计时的示例。只有一些 numpy 调用会得益于外部库的并行化。

9.4 寻找素数

接下来，我们会查看在一个大数值范围内测试素数。这是一个与估算 pi 不同的问题，因为工作负载会变化，这取决于你在数值范围中的位置，并且每一个数字检查都具有不可预测的复杂度。我们可以创建一个串行的例程来检测素数，接着给每个进程传递可能的因子集来做检查。要并行化这个问题是令人为难的，这意味着没有需要被共享的状态才行。

Multiprocessing 模块使控制工作负载变得容易，所以我们应该会调查该如何调制队列来使用（和误用！）计算资源，并且探索出一个简单的方法来稍稍更有效地来使用我们的资源。这意味着我们会看看负载平衡来设法有效地把可变复杂度的任务分配给我们的固定资源集。

如果我们有一个偶数的话，我们会使用一个稍加改进的来自本书前面章节的（请看第 9 页上的“理想计算模型 Python 虚拟机”）算法，请看例 9-4。

例 9-4　使用 Python 来寻找素数

```
def check_prime(n):
    if n % 2 == 0:
        return False
    from_i = 3
```

```
    to_i = math.sqrt(n) + 1
    for i in xrange(from_i, int(to_i), 2):
        if n % i == 0:
            return False
    return True
```

当用这种方式检测素数时，有多少工作负载的变化是我们看得到的？图 9-10 显示了当可能的素数 n 从 10000 增长到 1000000 时所增加的检测素数的时间开销。

大多数数字是非素数的，它们用一个点来描绘。有些可以花很少的代价检测到，然而另一些则需要检查很多因子。素数由一个 x 来描绘，并且形成了一个厚厚的深色带，它们是检测开销最大的。检测一个数字的时间开销随着 n 增长而增长，因为要检查的可能的因子的区间是以 n 的平方根来增长的。素数序列是不可预测的，所以我们无法决定一个数值区间的期望开销（我们可以估算它，但是不能确保它的负责度）。

对这张图表，我们对每个 *n* 测试了 20 次，并且采用了最快的结果来消除结果中的抖动。

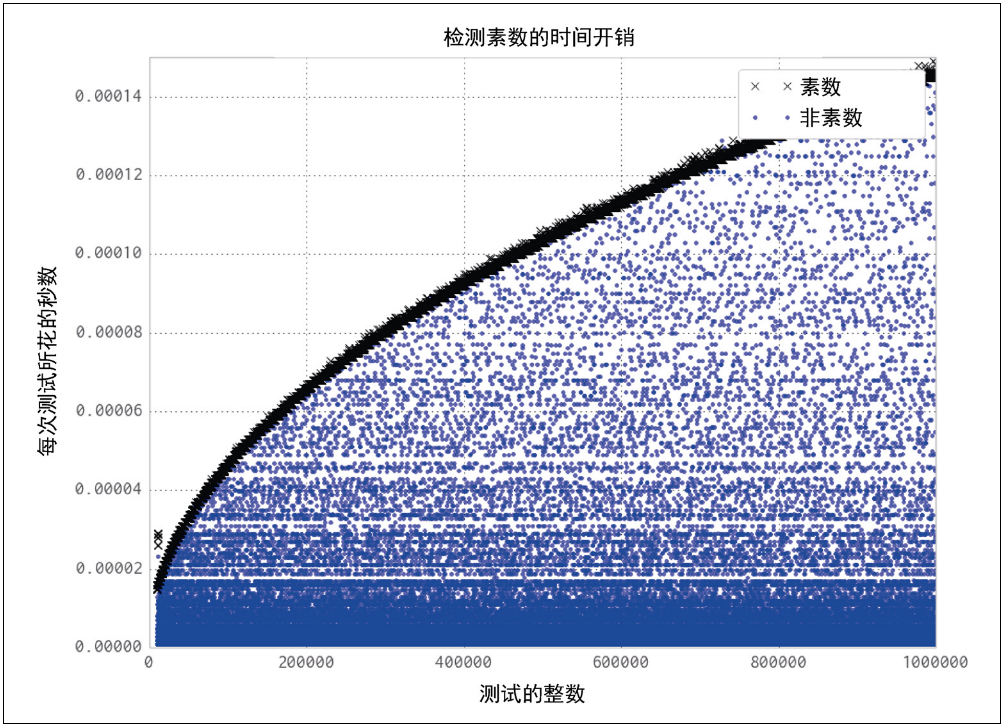


图 9-10　随着 *n* 增长，检测素数所需的时间

当我们给进程池分配工作时，我们可以指定要给每个工作者传递的工作量。我们可以均匀地划分所有工作并力求一次传递完，或者我们也可以创建很多工作块，当 CPU 空闲时就把它们传递出去。这是由 `chunksize` 参数来控制的。更大的工作块意味着更少的通信开销，而更小的工作块意味着对资源分配进行更多的控制。

对于我们的素数查寻器来说，一个单独的工作划片是一个由 `check_prime` 来检测的数字 `n`。`chunksize` 是 `10` 就意味着每一个进程处理一列 10 个整数，同时处理一列。

在图 9-11 中我们可以看到从 1（每个任务是一个单独的工作划片）到 64（每个任务是一列 64 个数字）之间变化的 `chunksize` 的效果。尽管有很多小任务给我们带来了最大的灵活性，它也强加了最大的通信开销。所有的 4 个 CPU 会有效地得以利用，但是当每一个任务和处理结果都经过一个单独的通信管道传输时，这个单独的信道就变成了一个瓶颈。如果我们把 `chunksize` 翻倍变成 2，我们的任务就以两倍快的速度得到了解决，因为我们在通信管道上有更少的竞争。我们可能会天真地假设通过增加 `chunksize`，我们会继续缩短执行时间。无论如何，就如你能在图表中所见的那样，我们将再次遇到一个回报减弱的点。

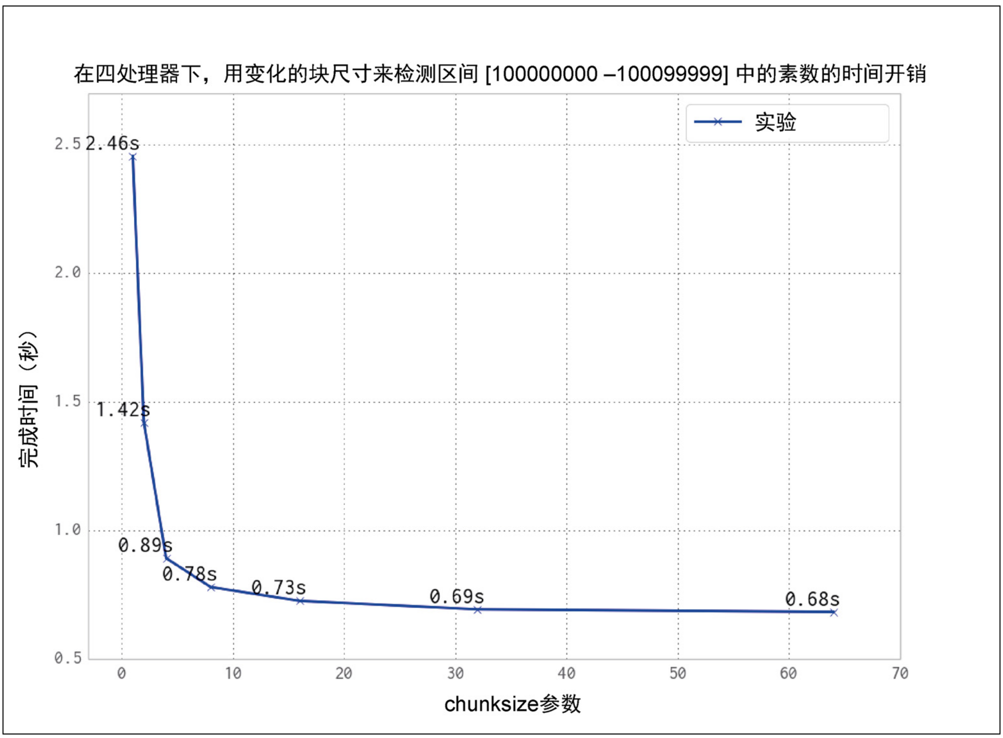


图 9-11　选择一个合理的 chunksize 值

我们可以继续增大 chunksize 直到开始发现表现变坏。在图 9-12 中我们扩展了块尺寸的区间，使得它们不但仅仅有小块，而且还有巨块。在区间的大端，最坏的结果显示为 1.31 秒，在那里我们已让 chunksize 变成了 50000——这意味着我们的 100000 项被划分成了两个工作块，使得两个 CPU 在整个扫描过程都空闲了下来。而使用具有 10000 项的 chunksize，我们创建了 10 个工作块，这意味着 4 个工作块将并行地运行两遍，接下来再运行剩余的 2 个工作块。这就让两个 CPU 在第三轮工作中空闲下来，是对资源的低效利用。

在这种情况下一个优化的解决方案是把所有数量的任务根据 CPU 的数量来划分。这是 multiprocessing 的默认行为，在图中显示为“默认”的黑点。

作为一个通用规则，默认的行为是明智的，只有当你期望看见一个真正的收益，并且对比默认行为确切地去证实你的假设时，才去调整它。

与蒙特卡罗的 pi 问题不同，我们的素数检测计算有着可变的复杂度——有时一个任务快速地结束了（偶数检测得最快），而有时一个数字很大，而且是素数（这要花费长得多的时间去检测）。

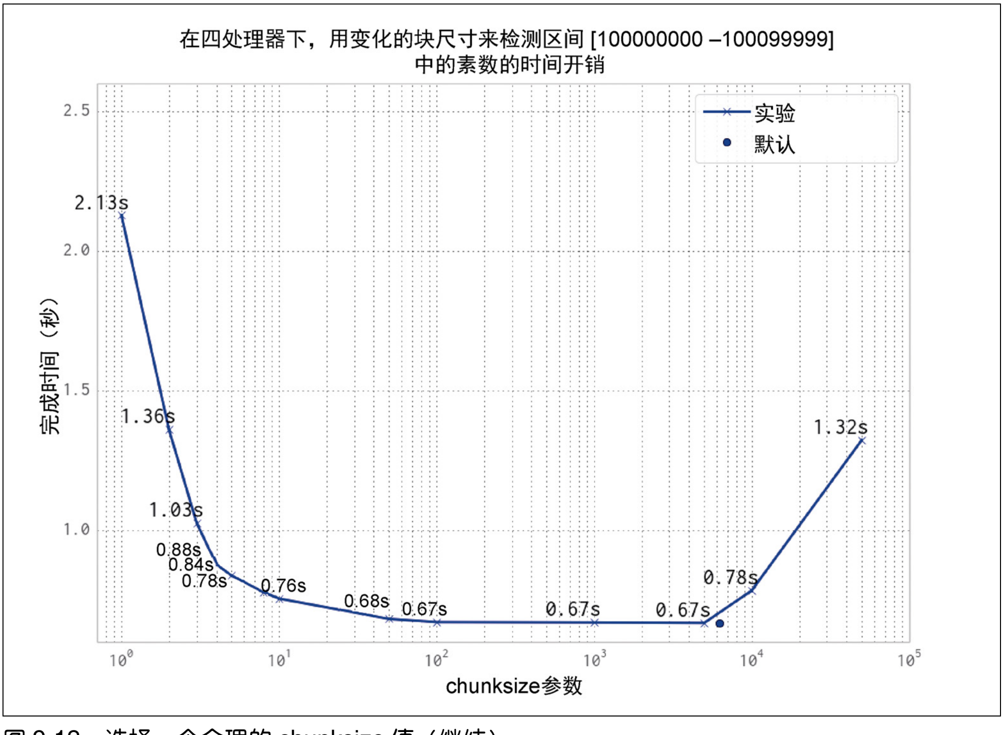


图 9-12　选择一个合理的 chunksize 值（继续）

如果我们随机化我们的工作序列，会发生什么呢？对于这个问题，我们压榨出了 2%的性能收益，就如你在图 9-13 中所见的那样。通过随机化，我们减少了序列中的最后任务比其他任务要花费更久时间运行，从而只让一个 CPU 处于活动状态的可能性。

就如我们更早的例子使用一个 `10000` 的 `chunksize` 所演示的那样，错配工作负载和可利用的资源会导致低效。在那种情况下，我们创建了三轮工作：开始的两轮使用了 100%的资源，而最后一轮仅使用了 50%。

图 9-14 展示了当我们错配工作块数量和处理器数量时，奇怪的现象会发生。错配会导致对资源利用不足。当仅有一个工作块被创建时，发生了最慢的整体运行时间。两个工作块让两个 CPU 没有得到利用，依次类推，只有当我们有 4 个工作块时，我们才使用上了所有的资源。但是如果我们增加了第 5 个工作块，那么我们会再次对资源利用不足——4 个 CPU 会工作于它们的块上，接着一个 CPU 将运行计算第 5 个块。

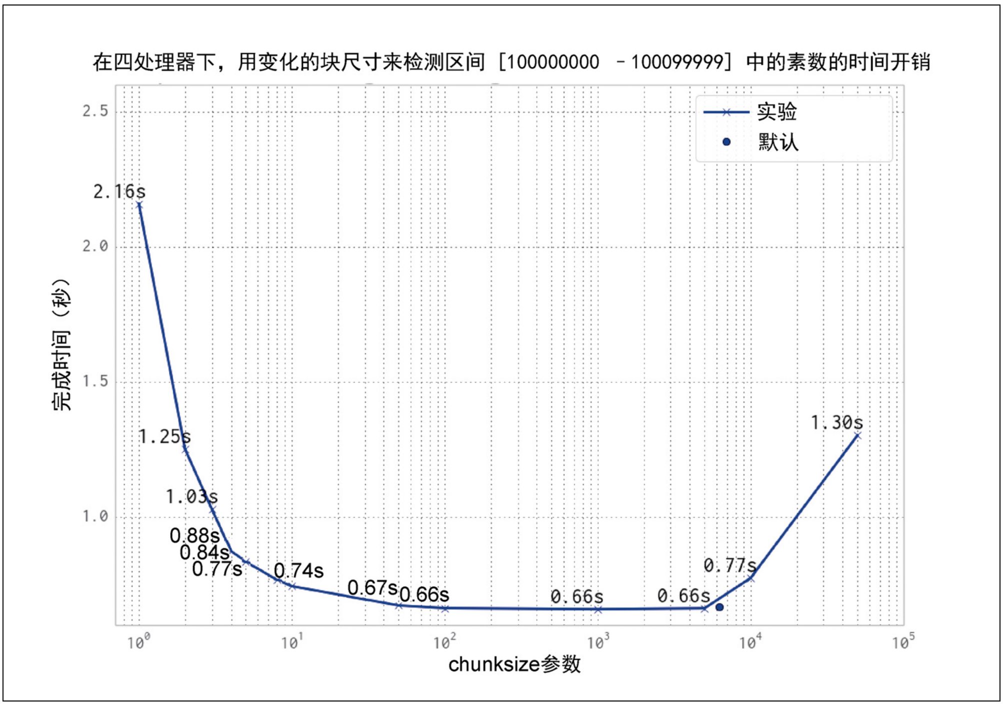


图 9-13　随机化任务队列

当我们增大了工作块数量时，我们看到低效程度减弱了——29 和 32 个工作块的运行时间差异大约是 0.01 秒。通用规则就是如果你的任务运行时是可变的，那就创建许多的小任务来有效使用资源。

有一些策略用来有效使用 multiprocessing 解决棘手的并行问题：

- 把你的工作拆分成独立的工作单元。
- 如果你的工作者所花的时间是可变的，那就考虑随机化工作序列（另一个例子就是处理大小可变的文件）。
- 对你的工作队列进行排序，这样首先处理最慢的任务可能是一个平均来说有用的策略。
- 使用默认的 chunksize，除非你已经验证了调节它的理由。
- 让任务数量与物理 CPU 数量保持一致（默认的 chunksize 再次为你考虑到了，尽管它默认会使用超线程，这样可能不会提供额外的性能收益）。

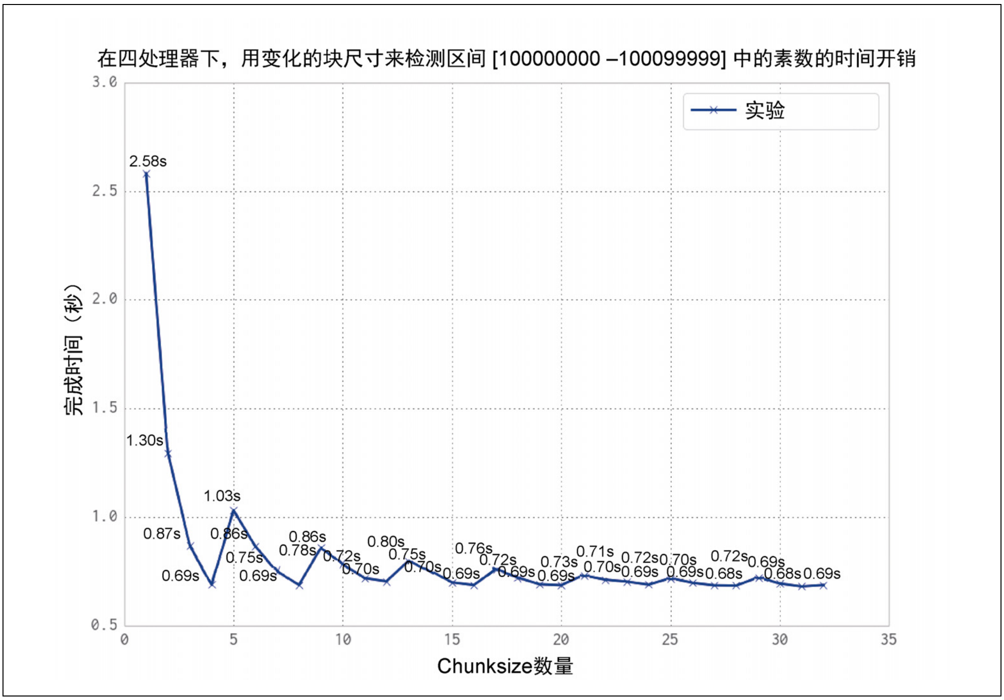


图 9-14　选择不合适的块数量的危险性

注意默认情况下，multiprocessing 会把超线程视作附加的 CPU。这意味着在 Ian 的笔记本电脑上，它会分配 8 个进程，而只有 4 个会真正跑出 100%的速度。多出的 4 个进程可能占用了珍贵的 RAM，却几乎没有提供任何额外的速度提升。

使用一个池，我们可以把一块预定义的工作事先在可用的 CPU 上拆分。然而如果我

们有动态的工作负载，尤其是如果我们有随时间而来的工作负载，这样做帮助就减少了。对于这种类型的工作负载，我们可能想要使用一个 Queue，在下一节会介绍。

备忘

如果你正工作于一个长期运行的科学问题，每个任务花费许多秒（或更长）来运行，那么你可能会想要检视下 Cael Varoquax 的 joblib。这个工具支持轻量级的流水线，它在 multiprocessing 之上设置，并且提供了一个更简单的并行接口、结果缓存和调试功能。

工作队列

multiprocessing.Queue 对象给我们非持久性的队列，能够在进程间传送任何可序列化（pickleable）的 Python 对象。当每个对象必须要被序列化（pickle）来传送，接着在消费者那里复原（伴随着一些加锁操作）时，它们就带来了一个开销。在下面的例子中，我们会看到这个代价是不可忽略的。无论如何，如果你的工作者正在处理更大的任务，那么通信开销可能是可接受的。

使用队列来工作相当简单。在这个例子中，我们会检测素数，通过消费一列候选数字并且把确认的素数发回一个 definite_primes_queue。我们会使用一个、两个、四个和八个进程来运行，并且证实后者都会比只运行一个单独的进程来检测相同的区间花费更长的时间。

Queue 带给我们使用原生的 Python 对象来执行许多进程间通信的能力。如果你正在用许多状态在对象间相互传递，这可能是有用的。然而，因为 Queue 缺乏持久性，你可能不想用它们来做在面临失效时需要鲁棒性的工作（例如，如果你断电了或者硬盘崩坏了）。

例 9-5 展示了 check_prime 函数。我们已经熟悉了基本的素数测试方法。我们运行于一个无限循环中，阻塞于（等待直到有可用的任务为止）possible_primes_queue.get() 上来从队列中消费一项任务。只有一个进程能够在同一时刻得到一项任务，因为 Queue 对象考虑到了同步存取。如果队列中没有任务，那么 .get() 就阻塞直到任务可用。当素数被找到时，它们被放回 definite_primes_queue 中来为父进程所消费。

例 9-5　使用两个队列来 IPC（进程间通信）

```
FLAG_ALL_DONE = b"WORK_FINISHED"
FLAG_WORKER_FINISHED_PROCESSING = b"WORKER_FINISHED_PROCESSING"

def check_prime(possible_primes_queue, definite_primes_queue):
```

```
    while True:
        n = possible_primes_queue.get()
        if n == FLAG_ALL_DONE:
            # flag that our results have all been pushed to the results queue
            definite_primes_queue.put(FLAG_WORKER_FINISHED_PROCESSING)
            break
        else:
            if n % 2 == 0:
                continue
            for i in xrange(3, int(math.sqrt(n)) + 1, 2):
                if n % i == 0:
                    break
        else:
            definite_primes_queue.put(n)
```

我们定义了两个标记：一个由父进程来表明没有可用的工作了，而第二个由工作者来确认它已经看到了毒药，并把自己关闭。第一个毒药也叫哨兵，因为它保证终结处理循环。

当处理工作队列和远程工作者时，使用像那样的标记有助于来记录毒药已送出，并且检查响应已在合理的时间窗口由子进程送出，从而表明它们正在关闭中。我们在这里不处理那个进程，但是增加一些时间记录对代码是一种相对简单的添加。这些标记的接收依据在调试期间能够被记入日志或者打印出来。

`Queue` 对象创建于例 9-6 中的 `Manager`。我们会使用熟悉的过程来构建一个 `Process` 对象列表，每一个包含了一个派生（fork）进程。两个队列被当作参数送出，`multiprocessing` 处理它们的同步。已经启动新进程后，我们就把一个任务列表移交给 `possible_primes_queue`，并且用毒药来终结每个进程。任务会以先进先出的顺序被消费，毒药留在最后面。在 `check_prime` 中，我们使用一个 `blocking.get()`，因为新进程不得不等待工作在队列中出现。既然我们使用了标记，我们就可能会增加一些工作，处理结果，接着通过增加更多的工作来遍历，并且通过在之后增加毒药来表明工作者生命的终结。

例 9-6　为 IPC（进程间通信）构建两个队列

```
if __name__ == "__main__":
    primes = []

    manager = multiprocessing.Manager()
    possible_primes_queue = manager.Queue()
    definite_primes_queue = manager.Queue()

    NBR_PROCESSES = 2
    pool = Pool(processes=NBR_PROCESSES)
    processes = []
```

```
    for _ in range(NBR_PROCESSES):
        p = multiprocessing.Process(target=check_prime,
                                    args=(possible_primes_queue,
                                    definite_primes_queue))
        processes.append(p)
        p.start()

    t1 = time.time()
    number_range = xrange(100000000, 101000000)

    # add jobs to the inbound work queue
    for possible_prime in number_range:
        possible_primes_queue.put(possible_prime)

    # add poison pills to stop the remote workers
    for n in xrange(NBR_PROCESSES):
        possible_primes_queue.put(FLAG_ALL_DONE)
```

为了消费结果，我们在例 9-7 中启动了另一个无限循环，在 definite_primes_queue 上使用一个 blocking.get()。如果 finished-processing 标记找到了，那么我们就对表明自己终结退出的进程计数。如果没有找到，那么我们就有一个新素数并把它添加到素数列表中去。当我们所有的进程已经表明自己终结退出时，我们就结束无限循环。

例 9-7　为 IPC（进程间通信）使用两个队列

```
processors_indicating_they_have_finished = 0
while True:
    new_result = definite_primes_queue.get() # block while waiting for results
    if new_result == FLAG_WORKER_FINISHED_PROCESSING:
        processors_indicating_they_have_finished += 1
        if processors_indicating_they_have_finished == NBR_PROCESSES:
            break
    else:
        primes.append(new_result)
assert processors_indicating_they_have_finished == NBR_PROCESSES

print "Took:", time.time() - t1
print len(primes), primes[:10], primes[-10:]
```

归因于序列化（pickle）和同步，使用 Queue 具有相当的开销。就如你在图 9-15 中所能看到的那样，使用一个更少的 Queue 的单进程解决方案明显要比使用两个或多个进程的要快。这种情况的原因就是我们的工作负载很轻——对于这个任务，通信开销占据了整体时间的大部分。使用 Queues，两个进程完成这个例子要比一个进程稍快一点，而四个或八个进程则比一个进程要更慢。

如果你的任务有较长的完成时间（至少相当多的秒数）和少量的通信，那么 Queue 的方式可能是正确的答案。你将不得不验证通信开销是否让这种方式足够有效。

你可能想知道如果我们移除了多余的一半工作队列（所有的偶数——这些偶数在 check_prime 中被很快地剔除了）会发生什么。把输入队列减半在每一种情况下都让我们的执行时间减半了，但是它还是没有战胜单进程非队列的例子！这有助于演示通信开销在这个问题中占主导因素。

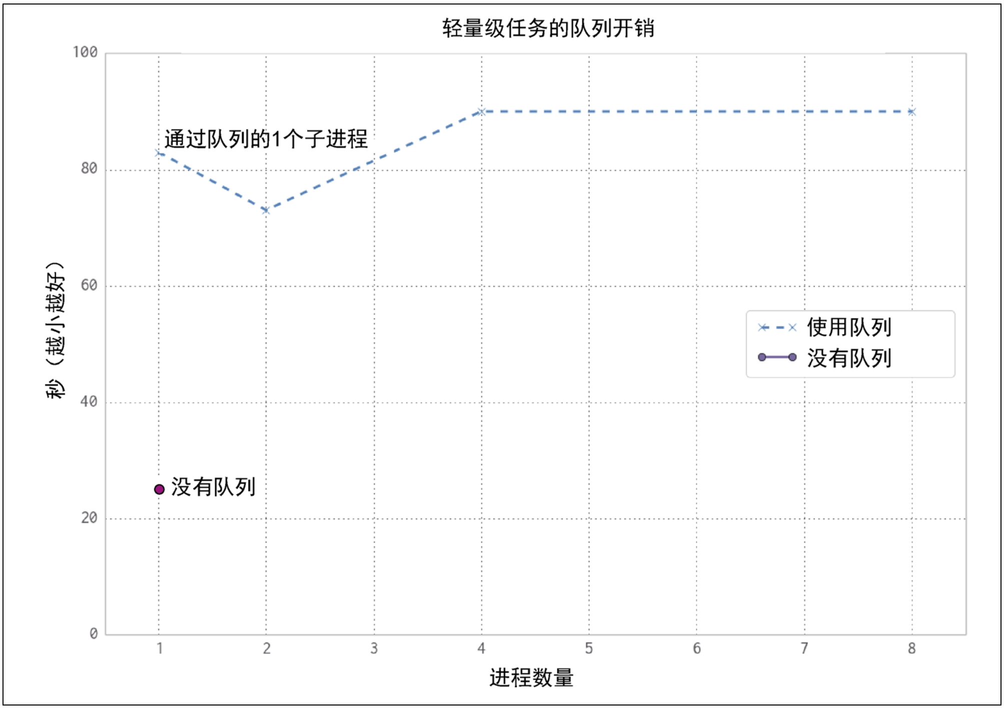


图 9-15　使用 Queue 对象的开销

异步地给 Queue 添加工作

通过给主进程增加一个 Thread，我们能够异步地给 possible_primes_queue 提供工作。在例 9-8 中，我们定义了一个 feed_new_jobs 函数：它就如我们在 __main__之前的工作设置例程那样，执行相同的工作，但是它却是在一个独立的线程中做的。

例 9-8　异步工作提供函数

```
def feed_new_jobs(number_range, possible_primes_queue, nbr_poison_pills):
    for possible_prime in number_range:
        possible_primes_queue.put(possible_prime)
    # add poison pills to stop the remote workers
    for n in xrange(nbr_poison_pills):
        possible_primes_queue.put(FLAG_ALL_DONE)
```

现在，在例 9-9 中，我们的__main__将使用 possible_primes_queue 来设置 Thread，接着在任何工作被发起前，继续移动到结果收集阶段。异步工作提供者能够从外部源中消费工作（例如，从一个数据库或者 I/O 密集型的通信中），而__main__线程来操作每一个处理后的结果。这意味着输入序列和输出序列不需要提前被创建，它们都能够被即时处理。

例 9-9　使用一个线程来设置一个异步工作提供者

```
if __name__ == "__main__":
    primes = []
    manager = multiprocessing.Manager()
    possible_primes_queue = manager.Queue()

    ...

    import threading
    thrd = threading.Thread(target=feed_new_jobs,
                            args=(number_range,
                                  possible_primes_queue,
                                  NBR_PROCESSES))
    thrd.start()

    # deal with the results
```

如果你想要健壮的异步系统，你几乎一定要看看成熟的外部库。gevent、tornado 和 Twisted 是强力的候选者，Python 3.4 的 tulip 是一个新的竞争者。我们在这里看的例子将让你起步，但是实际上相比对生产系统的作用，它们对很简单的系统和培训来说更有用。

备忘

另一个你可能想要调查的单机队列是 PyRes。这个模块使用了 Redis（在 9.5.5 节介绍过）来存储队列的状态。Redis 是一个非 Python 的数据存储系统，这意味着由 Redis 所持有的队列数据在 Python 之外是可读的，并且能够被非 Python 的系统所共享。

要非常注意，异步系统需要一个特殊级别的耐心——当你在调试时，你会在撕扯你的头发中结束。我们要建议：

- 应用“保持简单、愚蠢”的原则。
- 如果有可能尽量避免异步的自包含系统（就像我们的例子），因为它们的复杂度会增长，并且很快变得难以维护。
- 使用成熟的库，就像 gevent（在前一章中所描述的）那样给予你尝试和检验的方法来处理某些问题集。

而且，我们强烈建议使用一个外部的队列系统（例如，Gearman、0MQ、Celery、PyRes 或者 HotQueue）来给予你对队列状态的外部可视性。这需要更多的思考，但是归因于增加的调试效率和对生产系统的更好的系统可视性，可能会节省你的时间。

9.5 使用进程间通信来验证素数

素数是除了自己和 1 以外没有其他因子的数字。这正是绝大多数公因数是 2 的理由（每一个偶数都不能是素数）。然后，小素数（比如 3、5、7）成为了更大的非素数的公因子（比如，对应于 9、15、21）。

比如说给我们一个大数字，要我们验证它是否是素数。我们可能会有一个大范围的因子要去搜索。图 9-16 显示了一直到 10000000 的非素数的每一个因子出现的频率。小因子要比大因子出现的几率大得多，但是没有可预测的模式。

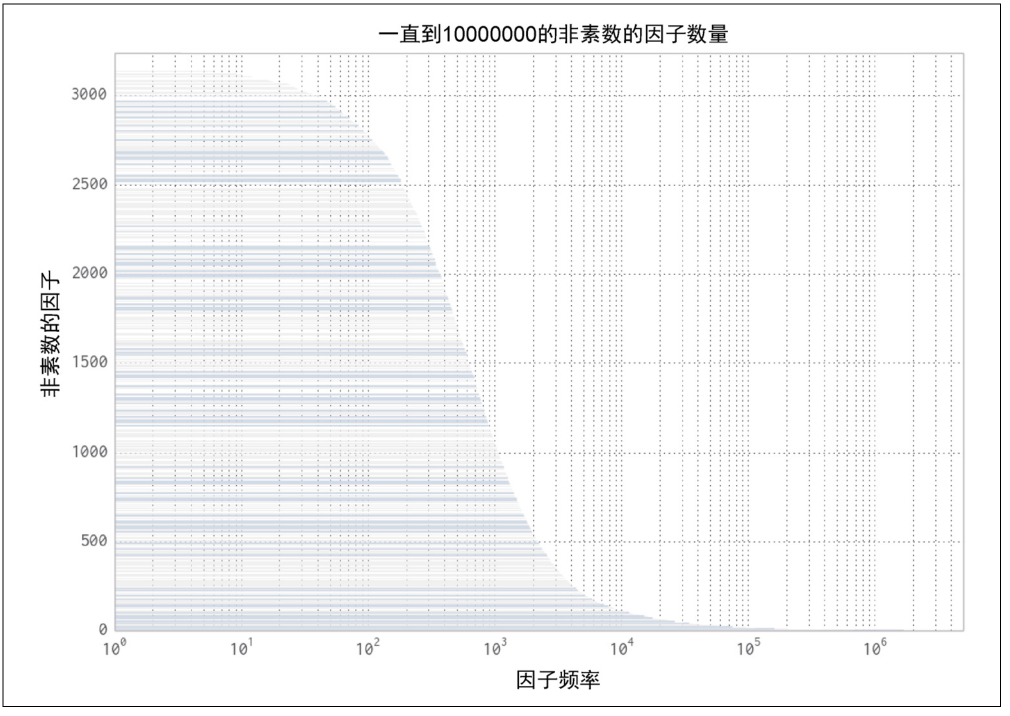


图 9-16　非素数的因子频率

让我们定义一个新的问题——假设我们有一个小数字集，我们的任务是有效地利用 CPU 资源来计算出每一个数字是否是素数，一时刻计算一个数字。可能我们会只

有一个大数字要检测。使用一个 CPU 来做检测不再有意义，我们想要在许多 CPU 之间协调工作。

在本节中我们会看看一些更大的数字，一个数字有 15 位数（digits），4 个数字有 18 位数（digits）：

- 小的非素数：112272535095295。
- 大的非素数 1：00109100129100369。
- 大的非素数 2：100109100129101027。
- 素数 1：100109100129100151。
- 素数 2：100109100129162907。

通过使用更小的非素数和一些更大的非素数，我们得到验证，我们所选择的过程不但在检测素数方面更快，而且在检测非素数方面不会变得更慢。我们会假设并不知道所给的数字大小或类型，所以我们想要对所有的使用情况来说可能是最快的结果。

协同带来了开销——同步数据和检查共享数据的开销可能是相当大的。我们在这儿会过一遍几种能够以不同的方式来做任务同步的方法。注意我们在这里没有涉及某些特殊化的消息传递接口（MPI），我们打算看看内置的模块和 Redis（很普遍）。

如果你想要使用 MPI，我们假设你已经知道正在做什么。MPI4PY 项目将会是一个起步的好地方。当很多进程要相互协同时，无论你是否有一台或多台机器，如果你想要控制延迟，它就是一个理想的技术。

在下列运行中，每个测试执行 20 遍，采用了最小的时间来展示对那种方法来说最快的速度。在这些例子中，我们要使用各种不同的技术来共享一个标记（通常是一个字节）。我们可能使用一个基本对象，比如一个 `Lock`，但是接着我们只可以共享一个比特的状态。我们会选择性地来向你展示如何共享一个原生类型，从而有可能共享更多富有表达力的状态（即使我们为这个例子不需要一个更有表达力的状态）。

我们必须强调共享状态倾向于让事情变得复杂化——你会轻易地在另一个撕扯头发的状态中结束。要小心并且要尝试让事情尽可能保持简单。可能有一种情况，开发者花在其他挑战上的时间超过了更低效的资源利用。

首先我们要讨论结果，接着我们要过一遍代码。

图 9-17 展示了第一种方法来尝试使用进程间通信来更快地检测素数。基准就是没有使用任何进程间通信的串行版本，每一个想要加速我们代码的尝试至少需要比它快。

Less Naïve Pool 版本具有一个可预测（并且良好）的速度。它是足够好了，从而相当难以被击败。不要忽略你搜索快速解决方案中的那些显而易见的方案——有时一个笨拙又足够好的解决方案就是你所需要的全部。

Less Naïve Pool 解决方案中的方式就是采用我们在检测中的数字，在可用的 CPU 之间均匀地划分可能的因子区间,接着把工作推送到每一个 CPU 上。如果任何 CPU 发现了一个因子，它就提早结束，但是它不会交流这个因子，其他 CPU 会继续完成它们那部分区间工作。这意味着对于一个 18 位数的数字来说（我们 4 个更大的数字的例子)，无论它是素数还是非素数，搜索时间都是相同的。

Redis 和 Manager 的解决方案在遇到检测更多的因子数来求素数时会更慢，归因于通信开销。它们使用一个共享标记来表明已经发现一个因子，应该放弃搜索。

Redis 让你不仅与其他 Python 进程共享状态,而且还与其他工具和其他机器来共享，甚至通过一个 web 浏览器接口来暴露状态（可能对远程监控有用）。Manager 是 `multiprocessing` 的一部分，它提供了一个高级的 Python 对象的同步集合（包括原生对象、`list` 和 `dict`）。

对于更大的非素数的情况，尽管有检查共享状态的开销，但是在提早通知已经发现一个因子所节省的搜索时间面前，显得相形见绌。

然而，对于素数的情况，没有办法来提早结束，因为不会发现因子，所以检查共享标记的开销将占主导地位。

图 9-18 显示了我们能够凭借一点点努力来得到一个明显更快的结果。Less Naïve Pool 的结果仍旧是我们的基准，但是 RawValue 和 MMap（内存映射）的结果比之前 Redis 和 Manager 的结果要快得多。真正的魔法来自于采用了最快的解决方案，并且执行了一些更不明显的代码运算来做出了一个接近最优的 MMap 解决方案——对于非素数，这个最终版本比 Less Naïve Pool 解决方案要快，对于素数几乎一样快。

在接下来的小节中，我们将过一遍在 Python 中使用 IPC（进程间通信）的各种各样的方式来解决我们的合作搜索问题。我们希望你会看到 IPC（进程间通信）是相当的简单，但是一般会伴随着一定的开销。

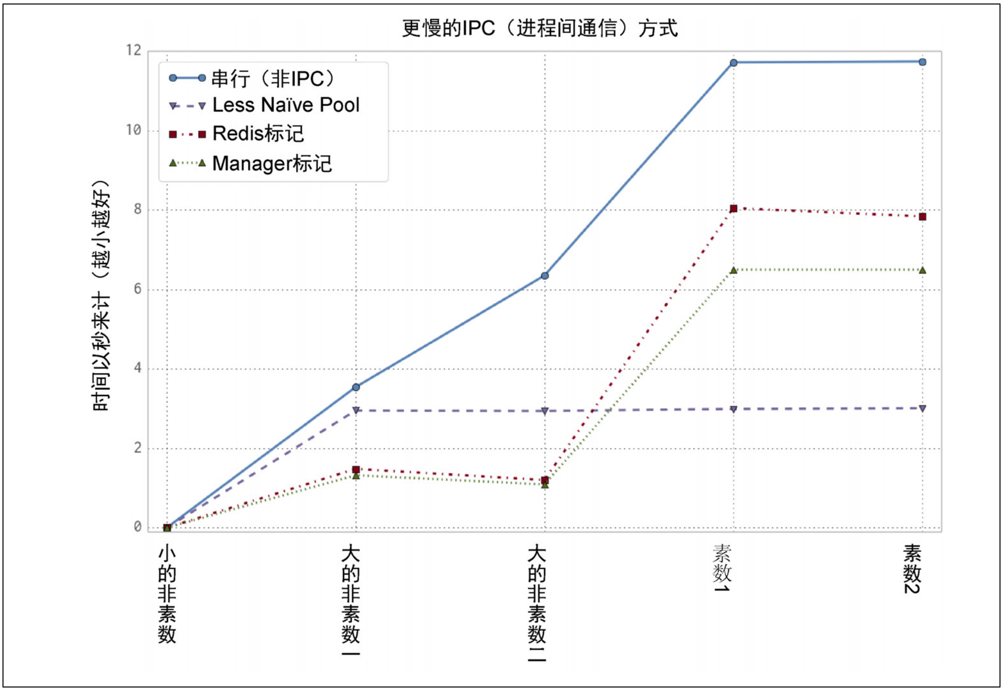


图 9-17　更慢的方式来使用 IPC（进程间通信）验证素数

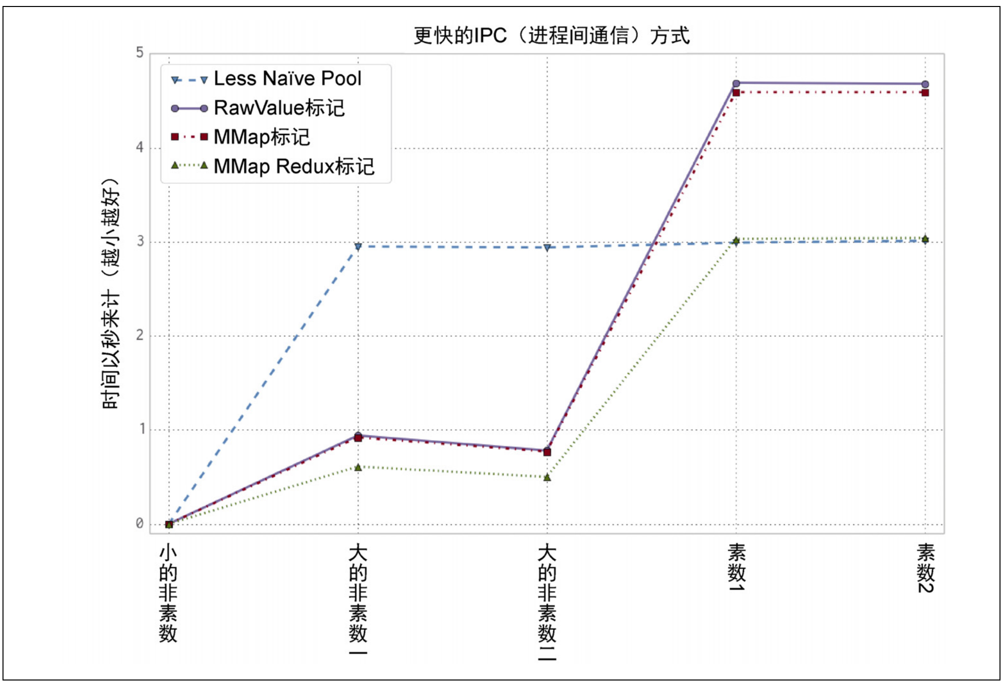


图 9-18　更快的方式来使用 IPC（进程间通信）验证素数

9.5.1　串行解决方案

我们将以与之前所使用的相同的串行因子检查的代码作为开始，再次展示于例 9-10 中。就如之前所要注意的那样，对于任何有大因子的非素数，我们可以更有效地并行搜索因子空间。一个串行清扫器还是会给我们一个有意义的基线，从而基于它来工作。

例 9-10　串行验证

```
def check_prime(n):
    if n % 2 == 0:
        return False
    from_i = 3
    to_i = math.sqrt(n) + 1
    for i in xrange(from_i, int(to_i), 2):
        if n % i == 0:
            return False
    return True
```

9.5.2　Naïve Pool 解决方案

Naïve Pool 解决方案使用一个 multiprocessing.Pool 来工作，类似于我们在 9.4 节所见到的“寻找素数”和在 9.3 节中使用 4 个派生（fork）进程的“使用多进程和多线程来估算 Pi”。我们有一个数字来检测它是否为素数，并且我们把可能的因子区间划分为四个子区间元组，再把它们发送到 Pool 中。

在例 9-11 中，我们使用了一个新方法，create_range.create（我们不会展示出来——它比较枯燥）来把工作空间分割成相同大小的区域，在 range_to_check 中的每一项是要在其中搜索的一对上下边界。对于第一个 18 位数的非素数（00109100129100369），使用 4 个进程，我们会有如下因子区间 ranges_to_check == [(3, 79100057), (79100057, 158200111), (158200111, 237300165), (237300165, 316400222)]（316400222 是 100109100129100369 的平方根加一）。在__main__中，我们首先创建了一个 Pool，接着 check_prime 通过一个 map 为每一个可能的素数来分割得到 ranges_to_check。如果结果是 False，那么我们已经发现了一个因子，就不会是一个素数。

例 9-11　Naïve Pool 解决方案

```
def check_prime(n, pool, nbr_processes):
    from_i = 3
    to_i = int(math.sqrt(n)) + 1
    ranges_to_check = create_range.create(from_i, to_i, nbr_processes)
    ranges_to_check = zip(len(ranges_to_check) * [n], ranges_to_check)
```

```
    assert len(ranges_to_check) == nbr_processes
    results = pool.map(check_prime_in_range, ranges_to_check)
    if False in results:
        return False
    return True

if __name__ == "__main__":
    NBR_PROCESSES = 4
    pool = Pool(processes=NBR_PROCESSES)
    ...
```

我们在例 9-12 中修改了之前的 check_prime，采用一个有上下边界的区间来做检测。在传递一个完整的可能的因子列表去检测的过程中，没有什么值，所以我们通过仅仅传递定义区间的两个数字来节省了时间和内存。

例 9-12　check_prime_in_range

```
def check_prime_in_range((n, (from_i, to_i))):
    if n % 2 == 0:
        return False
    assert from_i % 2 != 0
    for i in xrange(from_i, int(to_i), 2):
        if n % i == 0:
            return False
    return True
```

对于“小的非素数”的情况，通过 Pool 的验证时间是 0.1 秒，明显要比原来的串行解决方案中的 0.00002 秒时间更长。尽管有这一个更糟的结果，整体结果却得到了一个全面的速度提升。我们可能会接受认为出现这一个更慢的结果不是问题——但是如果我们可能得到许多更小的非素数来检测，那会怎么样呢？我们有能力证明能够避免这种减速，我们会看看接下来的 Less Naïve Pool 的解决方案。

9.5.3　Less Naïve Pool 解决方案

之前的解决方案在验证更小的非素数方面比较低效。对于任何更小的非素数（小于 18 个位数），有可能比串行方法要慢，这是因为发送分割的工作，以及不知道是否会找到一个非常小的因子（更可能的因子）所带来的开销。如果找到了一个小因子，那么进程还是得等到其他更大的因子搜索完成。

我们可以从在进程间发信号通知已经找到了一个小因子来开始，但是既然它发生得如此频繁，就会增加许多的通信开销。在例 9-13 中所呈现的解决方案是一种更实用的方法——为可能的小因子很快地执行一个串行检测，如果没有找到任何因子，那么启动一个并行搜索。在发起一个相对代价高昂的并行操作之前结合一个串行的预检是一种普遍的避免并行计算的部分开销的做法。

例 9-13　对于小的非素数的情况，改进了 Naïve Pool 的解决方案

```
def check_prime(n, pool, nbr_processes):
    # cheaply check high-probability set of possible factors
    from_i = 3
    to_i = 21
    if not check_prime_in_range((n, (from_i, to_i))):
        return False

    # continue to check for larger factors in parallel
    from_i = to_i
    to_i = int(math.sqrt(n)) + 1
    ranges_to_check = create_range.create(from_i, to_i, nbr_processes)
    ranges_to_check = zip(len(ranges_to_check) * [n], ranges_to_check)
    assert len(ranges_to_check) == nbr_processes
    results = pool.map(check_prime_in_range, ranges_to_check)
    if False in results:
        return False
    return True
```

对于我们的每一个测试数字，这种解决方案的速度等于或比原来的串行搜索要更快。这是我们的新基准。

重要的是，这种 Pool 方式给了我们面对素数检测场景的一种最优的情况。如果我们有素数，那么没有办法提早退出，我们不得不在可以退出前人工检查所有可能的因子。

没有最快的方式来检测完这些因子：任何增加复杂性的方法具有更多的指令，所以检测所有因子的情况会导致最多的指令要被执行。看看后面讲到的各种各样的 mmap 解决方案来讨论如何尽可能为素数得到接近于当前的这个结果。

9.5.4　使用 Manager.Value 作为一个标记

multiprocessing.Manager()让我们在进程之间共享更高级的 Python 对象作为被管理的共享对象，更低级的对象封装于代理对象中。封装和安全性会付出速度的代价，但也会提供巨大的灵活性。你既可以共享更低级的对象（例如，整数和浮点数），也可以共享列表和字典。

在例 9-14 中我们创建了一个 Manager，接着创建了一个 1 字节(character)的 manager.Value(b”c”, FLAG_CLEAR)标记。如果你想要共享字符串或数字，你可能会创建任何的 ctypes 原语（和 array.array 原语一样）。

注意 FLAG_CLEAR 和 FLAG_SET 被分配给 1 字节（各自是 b’0’和 b’1’）。我们选择使用以 b 开头的作为显式（如果留作一个隐式的字符串，它可能默认成一个 Unicode 或者字符串对象，这要取决于你的环境和 Python 版本）。

现在我们能够在我们所有的进程之间做标记表明一个因子已经找到了，所以搜索能够被提早取消。困难之处就是要平衡读取标记的开销相对于可能会挽救下来的速度。因为标记被同步了，我们就不用太频繁地检测了——这增加了更多的开销。

例 9-14　把一个 Manager.Value 对象作为一个标记

```
SERIAL_CHECK_CUTOFF = 21
CHECK_EVERY = 1000
FLAG_CLEAR = b'0'
FLAG_SET = b'1'
print "CHECK_EVERY", CHECK_EVERY

if __name__ == "__main__":
    NBR_PROCESSES = 4
    manager = multiprocessing.Manager()
    value = manager.Value(b'c', FLAG_CLEAR) # 1-byte character
    ...
```

check_prime_in_range 现在会留意共享标记，并且例程会检查看看一个素数是否已经被其他的进程所识别出。即使我们已经开始了并行搜索，我们也必须如例 9-15 中所示的那样在开始做串行检测前清除标记。在已经完成串行检测之后，如果我们没有找到因子，那么我们就知道标记肯定还是为否。

例 9-15　使用一个 Manager.Value 来清理标记

```
def check_prime(n, pool, nbr_processes, value):
    # cheaply check high-probability set of possible factors
    from_i = 3
    to_i = SERIAL_CHECK_CUTOFF
    value.value = FLAG_CLEAR
    if not check_prime_in_range((n, (from_i, to_i), value)):
        return False

    from_i = to_i
    ...
```

我们该以怎样的频率来检查共享标记呢？每一次检查都有开销，既是因为我们给紧致的内循环增加了更多的指令，又因为做检查需要在共享变量上创建一把锁，这就增加了更多的开销。我们所选择的解决方案就是每 1000 次循环检查一次标记。每次我们检查看看 value.value 是否已经被设置成了 FLAG_SET，如果已经设置了，我们就退出搜索。如果进程在搜索中找到了一个因子，那么它就设置 value.value = FLAG_SET 并退出（请看例 9-16）。

例 9-16　把 Manager.Value 对象作为一个标记来传递

```
def check_prime_in_range((n, (from_i, to_i), value)):
    if n % 2 == 0:
```

```
            return False
    assert from_i % 2 != 0
    check_every = CHECK_EVERY
    for i in xrange(from_i, int(to_i), 2):
        check_every -= 1
        if not check_every:
            if value.value == FLAG_SET:
                return False
            check_every = CHECK_EVERY

        if n % i == 0:
            value.value = FLAG_SET
            return False
    return True
```

在这个代码中的 1000 次迭代检查使用一个 `check_every` 本地计数器来执行。它证明了这种方法尽管可读性好，但是只得到了次优的速度。在本节结束前，我们会用一个可读性较差但是明显更快的方式来取代它。

你可能会对我们检查共享标记的总次数产生好奇。在两个大素数的情况下，我们使用 4 个进程对标记检查了 316405 次（我们在下面所有的例子中检查了这样的许多次）。既然每次检查有加锁的开销，这个代价真的合乎常理。

9.5.5　使用 Redis 作为一个标记

Redis 是一个在内存中的键/值对存储引擎。它提供了自己的锁，每一个运算是原子性的，所以我们不必担心在 Python 内部使用锁（或者从任何其他的接口语言）。

通过使用 Redis，我们让数据存储变得和语言无关——任何与 Redis 有接口的语言或工具都能以一种可兼容的方式来共享数据。你可以轻易地在 Python、Ruby、C++和 PHP 之间平等地共享数据。你可以在本地机器或者网络上共享数据。为了共享给其他机器，你所需要做的全部事情就是改变 Redis 默认只在 `localhost` 共享的配置。

Redis 让你存储：

- 字符串列表。
- 字符串集合。
- 有序的字符串集合。
- 字符串散列。

Redis 在 RAM 中存储一切，把快照存储到磁盘（有选择地使用日志），并支持到一个实例集群的主从复制。用 Redis 的一种可能性就是使用它来在集群中共享工作负载和其他机器的读写状态，而 Redis 扮演着一个快速的中心化的数据仓库角色。

我们能够把一个标记作为一个文本字符串来读写（在 Redis 中的所有值都是字符串），就像之前我们已经使用过的 Python 标记的方式一样。我们创建了一个 `StrictRedis` 接口作为一个全局对象，和其他外部的 Redis 服务器来通信。我们可以在 `check_prime_in_range` 内部创建一个新连接，但是这会更慢，并且能消耗可利用的数量有限的 Redis 句柄。

我们使用类似字典的存取方式来与 Redis 服务器通信。我们能使用 `rds[SOME_KEY] = SOME_VALUE` 来设置一个值，并且能使用 `rds[SOME_KEY]` 来读回字符串。

例 9-17 与之前的 `Manager` 例子很类似——我们使用 Redis 来作为本地 `Manager` 的代替物。结果它表现出类似的存取开销。你应该注意到 Redis 支持其他（更复杂）的数据结构。它是一个强大的存储引擎，我们用这个例子只是用它来共享一个标记。我们鼓励你自己去熟悉它的特点。

例 9-17　为我们的标记使用一个外部的 Redis 服务器

```
FLAG_NAME = b'redis_primes_flag'
FLAG_CLEAR = b'0'
FLAG_SET = b'1'

rds = redis.StrictRedis()

def check_prime_in_range((n, (from_i, to_i))):
    if n % 2 == 0:
        return False
    assert from_i % 2 != 0
    check_every = CHECK_EVERY
    for i in xrange(from_i, int(to_i), 2):
        check_every -= 1
        if not check_every:
            flag = rds[FLAG_NAME]
            if flag == FLAG_SET:
                return False
            check_every = CHECK_EVERY

        if n % i == 0:
            rds[FLAG_NAME] = FLAG_SET
            return False
    return True

def check_prime(n, pool, nbr_processes):
```

```
# cheaply check high-probability set of possible factors
from_i = 3
to_i = SERIAL_CHECK_CUTOFF
rds[FLAG_NAME] = FLAG_CLEAR
if not check_prime_in_range((n, (from_i, to_i))):
    return False
...
if False in results:
    return False
return True
```

为了确认数据存储在了这些 Python 实例外部，我们可以在命令行中调用 redis-cli，就如在例 9-18 中的那样，还可以得到存储在键 redis_primes_flag 中的值。你会注意到返回项是一个字符串（不是一个整数）。从 Redis 返回的所有值都是字符串，所以如果你想要用 Python 来操控它们，你必须首先把它们转换成一个合适的数据类型。

例 9-18　redis-cli 例子

```
$ redis-cli
redis 127.0.0.1:6379> GET "redis_primes_flag"
"0"
```

一个强有力的赞成使用 Redis 来共享数据的论据就是它存活于 Python 的世界之外——你的团队中的非 Python 开发者能理解它，还有许多为 Redis 的工具。在阅读（但没必要运行和调试）你的代码和跟踪发生了什么事情时，它们能够看到状态。从团队速度的角度来讲，尽管存在使用 Redis 的通信开销，这可能对你是一个巨大的胜利。尽管 Redis 对于你的项目有额外的依赖性，但你应该意识到它是一个非常普遍的部署工具，并且易于调试和理解。把它视作一个强大的工具添加到你的武器库中去。

Redis 有许多配置项。它默认使用一个 TCP 接口（正是我们所使用的），尽管基准文档提到套接字（socket）可能会快很多。它也陈述了尽管 TCP/IP 让你在不同类型的 OS 上跨网络共享数据，其他配置选项可能会更快（但是也受限于你的通信开销）：

> 当服务器和客户端基准程序运行于相同的盒子上时，既可以使用 TCP/IP 回路，也可以使用 UNIX domain socket。这取决于平台，但是 UNIX domain sockets 相比 TCP/IP 回路，能够达到大约超过 50%的吞吐量（在 Linux 实例上）。redis 基准的默认行为是使用 TCP/IP 回路。当管道被高负载使用时（例如，长管道），UNIX domain socket 相比 TCP/IP 回路的性能收益会倾向于减弱。
>
> ——Redis 文档

9.5.6 使用 RawValue 作为一个标记

multiprocessing.RawValue 是一个围绕 ctypes 字节块的薄薄的包装器。它缺少同步原语，所以几乎不会阻碍我们搜寻最快的方式来为进程间设置标记。它几乎和接下来的 mmap 例子一样快（只是因为受阻于多出来的一些指令，所以运行要慢一点）。

我们再次能够使用任何 ctypes 原语，也有一个 RawArray 选项来共享基本对象数组（变现类似于 array.array）。RawValue 避开了任何加锁——它使用起来更快，但是你不会得到原子操作。

一般来说，如果你避开了 Python 在进程间通信（IPC）期间所提供的同步，你会遭受挫折（再一次回到撕扯头发的境地）。无论如何，在这个问题中，不介意一个或多个进程同时设置标记——标记只会在一个方向上切换，每过一段时间读取它时，就会知道搜索是否能被取消。

因为我们在并行搜索期间从不重置标记状态，我们就不必同步。注意这样可能不会适用于你的问题。如果你避开了同步，请确保你以正确的理由来这样做。

如果你想要像更新一个共享计数器那样来做，请看一下 Value 的文档，并以 value.get_lock()来使用一个上下文管理器，因为对一个 Value 隐式加锁不允许原子操作。

这个例子看上去与之前的 Manager 例子很类似。仅有的差异就是在例 9-19 中我们创建了 RawValue 作为一个 1 字节（byte）的标记。

例 9-19 创建并传递一个 RawValue

```
if __name__ == "__main__":
    NBR_PROCESSES = 4
    value = multiprocessing.RawValue(b'c', FLAG_CLEAR) # 1-byte character
    pool = Pool(processes=NBR_PROCESSES)
    ...
```

使用受管理的值和原始值的灵活性正是在 multiprocessing 中为共享数据所做的干净设计而得到的收益。

9.5.7 使用 mmap 作为一个标记

最后，我们得到了共享字节的最快的方式。例 9-20 展示了使用 mmap 模块的内存映射（共享内存）的解决方式。共享内存块中的字节没有被同步，它们几乎没有什么开销。它们表现得就像一个文件——在这种情况下，它们就是具有类似文件接口

的一个内存块。我们必须要定位到一个位置上并连续地进行读或写。典型情况下，mmap 被用来给出一个较大文件的一个短小的视图（内存映射），但是在我们的用例中，不是指明一个文件号作为第一个参数，而是传递-1 来表明我们想要一个匿名的内存块。我们也能够指明我们是否想要只读或只写存取权限（我们想要全部权限，这是默认的权限）。

例 9-20　由 mmap 来使用一个共享内存标记

```
sh_mem = mmap.mmap(-1, 1) # memory map 1 byte as a flag


def check_prime_in_range((n, (from_i, to_i))):
    if n % 2 == 0:
        return False
    assert from_i % 2 != 0
    check_every = CHECK_EVERY
    for i in xrange(from_i, int(to_i), 2):
        check_every -= 1
        if not check_every:
            sh_mem.seek(0)
            flag = sh_mem.read_byte()
            if flag == FLAG_SET:
                return False
            check_every = CHECK_EVERY

        if n % i == 0:
            sh_mem.seek(0)
            sh_mem.write_byte(FLAG_SET)
            return False
    return True


def check_prime(n, pool, nbr_processes):
    # cheaply check high-probability set of possible factors
    from_i = 3
    to_i = SERIAL_CHECK_CUTOFF
    sh_mem.seek(0)
    sh_mem.write_byte(FLAG_CLEAR)
    if not check_prime_in_range((n, (from_i, to_i))):
        return False

    ...
    if False in results:
        return False
    return True
```

mmap 支持很多方法，能够被用来在它所代表的文件中腾挪（包括 find、readline 和 write）。我们正在以最基本的方式来使用它——我们在开始每一个读或写之前

都定位到内存块的起点处,既然我们只共享一个字节,我们显示地使用 read_byte 和 write_byte。

没有加锁和解释数据的 Python 开销，我们直接用操作系统来处理字节，所以这是最快的通信方式。

9.5.8 使用 mmap 作为一个标记的终极效果

尽管前面 mmap 的结果整体上已经是最好了,我们还是情不自禁地想到对于出现素数的代价最高的情况下，我们只能回到 Naïve Pool 的结果。目标就是去接受无法从内循环中提早退出的现实，并且最小化任何没有直接关联的开销。

本节呈现了一种稍微复杂一点的解决方案。尽管这个 mmap 的结果还是最快的，但是能够对其他我们已见到的基于标记的方法做出相同的修改。

在之前的例子中，我们已经使用了 CHECK_EVERY。这意味着我们可以有 check_next 局部变量来跟踪、减少，并使用布尔测试——每一个运算给每一次迭代增添了一点额外的时间。在验证一个大素数的情况下，这种额外的管理开销发生了超过 300000 次。

在例 9-21 中展示的第一个优化就是认识到我们可以用一个向前看（look-ahead）的值取代递减的计数器，接着我们只要在内循环中做一个布尔变量的比较即可。这样就移除了递减，归因于 Python 的解释风格，递减相当慢。这个优化用 CPython2.7 在这个测试中可以工作，但是它不可能在更智能的编译器（例如，PyPy 或 Cython）中提供任何收益。当检测我们其中一个大素数时，这个步骤节省了 0.7 秒的时间。

例 9-21 开始优化掉我们代价高昂的逻辑

```
def check_prime_in_range((n, (from_i, to_i))):
    if n % 2 == 0:
        return False
    assert from_i % 2 != 0
    check_next = from_i + CHECK_EVERY
    for i in xrange(from_i, int(to_i), 2):
        if check_next == i:
            sh_mem.seek(0)
            flag = sh_mem.read_byte()
            if flag == FLAG_SET:
                return False
            check_next += CHECK_EVERY

        if n % i == 0:
            sh_mem.seek(0)
            sh_mem.write_byte(FLAG_SET)
```

```
            return False
    return True
```

我们也能够通过把循环展开成两阶段过程的方式来整体替换计数器所代表的逻辑，就如在例 9-22 中所显示的那样。首先，外循环涉及期望区间，但是以逐步的方式作用在 CHECK_EVERY 上。其次，一个新的内循环取代了 check_every 逻辑——它检查局部因子区间，接着就结束了。这就等价于 if not check_every:test。我们用之前 sh_mem 的逻辑跟踪它来检查早退标记。

例 9-22　优化掉我们代价高昂的逻辑

```
def check_prime_in_range((n, (from_i, to_i))):
    if n % 2 == 0:
        return False
    assert from_i % 2 != 0
    for outer_counter in xrange(from_i, int(to_i), CHECK_EVERY):
        upper_bound = min(int(to_i), outer_counter + CHECK_EVERY)
        for i in xrange(outer_counter, upper_bound, 2):
            if n % i == 0:
                sh_mem.seek(0)
                sh_mem.write_byte(FLAG_SET)
                return False
        sh_mem.seek(0)
        flag = sh_mem.read_byte()
        if flag == FLAG_SET:
            return False
    return True
```

这对速度的影响是巨大的。对于非素数的情况甚至提高得更多，但是更重要的是，我们的素数检测已经几乎和 Less Naïve Pool 的版本一样快了(现在只慢了 0.05 秒)。在我们已经用进程间通信做了很多额外工作的前提下，这是非常令人感兴趣的结果。然而，要注意的是，它是针对 CPython 的，当通过编译器运行时不可能得到任何收益。

我们甚至能够走得更远 (但坦率地说，这有点儿蠢)。查找不在局部域中声明的变量开销有点高。我们可以创建全局 FLAG_SET 和频繁使用的.seed()和.read_byte()方法的的局部引用来避免代价更高的查找。然而，结果代码（例 9-23）的可读性甚至比前面的还差，我们真的不推荐你这样做。当检测更大的素数时，这个最终结果比 Less Naïve Pool 版本要慢上 1.5%。在我们对非素数得到了 4.8 倍加速的前提下，我们可能采用这个例子来看看它究竟（应该）能运行多快。

例 9-23　打破“不要伤害团队速度”的规则来竭力得到一个额外的加速

```
def check_prime_in_range((n, (from_i, to_i))):
    if n % 2 == 0:
        return False
```

```
    assert from_i % 2 != 0
    FLAG_SET_LOCAL = FLAG_SET
    sh_seek = sh_mem.seek
    sh_read_byte = sh_mem.read_byte
    for outer_counter in xrange(from_i, int(to_i), CHECK_EVERY):
        upper_bound = min(int(to_i), outer_counter + CHECK_EVERY)
        for i in xrange(outer_counter, upper_bound, 2):
            if n % i == 0:
                sh_seek(0)
                sh_mem.write_byte(FLAG_SET)
                return False
        sh_seek(0)
        if sh_read_byte() == FLAG_SET_LOCAL:
            return False
    return True
```

使用手工循环展开和创建全局对象的局部引用的行为是愚蠢的。整体上，它让代码变得更难以理解，从而受困于更低的团队速度，事实上这是编译器的工作（例如，一个类似 PyPy 的 JIT 编译器或者一个类似 Cython 的静态编译器）。

人们不应该做这种类型的操作，因为它是很脆弱的。我们用 Python 3+没有测出这种优化方式，而且我们不想要——我们不真心期待这些递进的提高在另外的 Python 版本（当然不是在不同的实现上，类似 PyPy 或 IronPython）上能工作。

我们向你展示过了，所以你知道它或许是可能的，并且提醒你保持理智，你真的应该让编译器来为你负责这种类型的工作。

9.6 用 multiprocessing 来共享 numpy 数据

当工作于大 `numpy` 数组时，你一定会想知道是否能够在进程间为读写存取来共享数据，而不用拷贝数据。这是可能的，尽管有一点烦琐。为了这个演示的灵感，我们要感谢 StackOverflow 的用户 pv。

警告

不要使用这个方法来重建 BLAS、MKL、Accelerate 和 ATLAS 的行为。这些库都用它们的原语来支持 multiprocessing，可能它们要比你所创建的任何新例程更好调试。尽管它们会需要一些配置来开启 multiprocessing 支持，但是在你投入时间（也失去了调试的时间！）来写你自己的代码之前，看看这些库是否能够给你带来免费的速度提升是明智的。

在进程间共享一个大矩阵有几个收益：

- 只有一个拷贝意味着没有浪费 RAM。
- 不浪费时间来拷贝大块内存。
- 你得到了在进程间共享部分结果的可能性。

回想起 9.3.3 节中使用 numpy 来估算 pi 的演示，我们出现问题，那就是随机数生成是一个串行过程。在这里，我们可以想象派生（fork）进程共享了一个大数组，每一个进程使用不同种子的随机数生成器用随机数来填充数组的一个区域，因此生成完一个大随机块可能要比用一个单独的进程要快。

为了验证它，我们修改了接下来的演示代码来以一个串行过程创建一个大随机矩阵（10000 乘 80000 个元素），再把矩阵拆分成 4 段来并行调用 random（在这两种情况下，同一时间处理一行）。串行过程花费 15 秒，并行版本花费 4 秒。回去参考 9.3.2 节来理解并行化随机数生成器的一些危险性。

在本节的余下部分，我们会使用一个简化的版本来演示一点，而剩下的容易验证。

在图 9-19 中，你能看见 Ian 的笔记本电脑上的 htop 输出。它显示了 4 个子进程的父进程（PID 是 11268），所有这 5 个进程共享了一个单独的 10000 × 80000 个元素的 double 型的 numpy 数组。这个数组的一份拷贝耗费 6.4GB，而笔记本电脑只有 8GB——你能看见在 htop 中通过进程计量器显示出 Mem 读取最多占用了 7491MB 的 RAM。

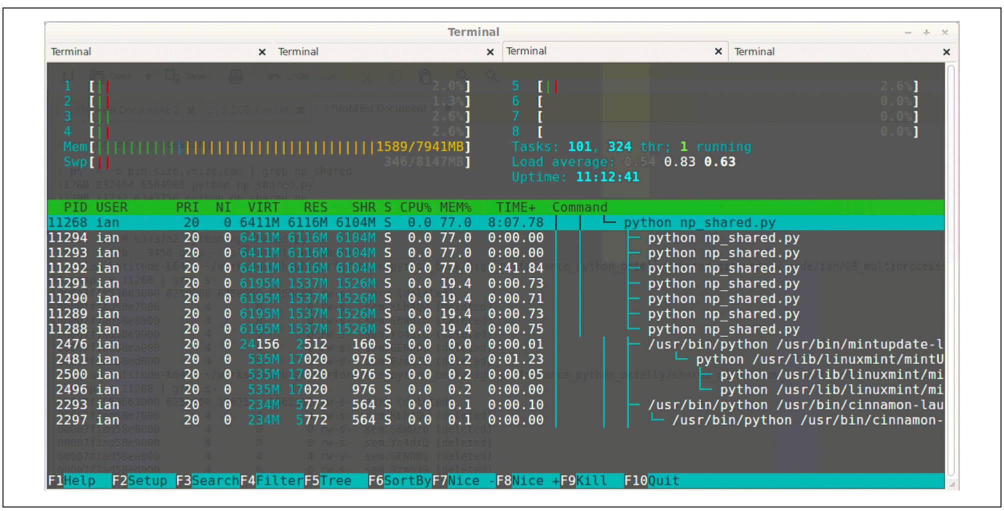


图 9-19　htop 显示了 RAM 和 swap 使用

为了理解这个演示，我们会首先过一遍控制台的输出，接着我们会看看代码。在例 9-24 中，我们启动了父进程：它分配了一个 6.4GB 的 double 型的 10000 × 80000 维的数组，以零值来填充。10000 行会作为索引传递给工作者函数，工作者将依次在每列 80000 项上操作。已经分配了数组，我们就用生命、宇宙，以及万物的答案（42！）来填充。我们能够在工作者函数中测试我们正接收着这个修改过的数组以及一个非填充 0 的版本来确认这份代码的表现和预期的一样。

例 9-24　设置共享数组

```
$ python np_shared.py
Created shared array with 6,400,000,000 nbytes
Shared array id is 20255664 in PID 11268
Starting with an array of 0 values:
[[ 0. 0. 0. ..., 0. 0. 0.]


 ...,
 [ 0. 0. 0. ..., 0. 0. 0.]]

Original array filled with value 42:
[[ 42. 42. 42. ..., 42. 42. 42.]
 ...,
 [ 42. 42. 42. ..., 42. 42. 42.]]
Press a key to start workers using multiprocessing...
```

在例 9-25 中，我们已经启动了 4 个进程工作于这个共享数组。不做任何的数组拷贝，每个进程着眼于相同的大内存块，并且每个进程有不同的索引集来工作。工作者每隔几千行输出当前的索引和它的 PID，所以我们可以观察它的行为。工作者的任务是琐碎的——它会检查当前元素还是设置在默认值（所以我们知道没有其他进程修改过它），接着它会用当前 PID 覆盖掉这个值。一旦工作者完成了，我们就回到父进程，再次打印数组。这次，我们看见它填充了 PID 而不是 42。

例 9-25　在共享数组上运行 worker_fn

```
worker_fn: with idx 0
 id of shared_array is 20255664 in PID 11288
worker_fn: with idx 2000
 id of shared_array is 20255664 in PID 11291
worker_fn: with idx 1000
 id of shared_array is 20255664 in PID 11289
...
worker_fn: with idx 8000
 id of shared_array is 20255664 in PID 11290

The default value has been over-written with worker_fn's result:
[[ 11288. 11288. 11288. ..., 11288. 11288. 11288.]
 ...,
 [ 11291. 11291. 11291. ..., 11291. 11291. 11291.]]
```

最后，在例 9-26 中我们使用了一个计数器来确认在数组中的每个 PID 的频率。因为工作是均匀划分的，我们期待 4 个 PID 中的每一个表示相等的次数。在我们的 800000000 个元素的数组中，我们看到了 4 组，每组 200000000 个 PID。使用 PrettyTable 来呈现表格输出。

例 9-26　在共享数组上验证结果

```
Verification - extracting unique values from 800,000,000 items
in the numpy array (this might be slow)...
Unique values in shared_array:
+---------+-----------+
| PID     | Count     |
+---------+-----------+
| 11288.0 | 200000000 |
| 11289.0 | 200000000 |
| 11290.0 | 200000000 |
| 11291.0 | 200000000 |
+---------+-----------+
Press a key to exit...
```

已经完成了，现在程序退出，数组被删除。

我们能够在 Linux 下用 ps 和 pmap 来查看每个进程内部。例 9-27 显示了调用 ps 的结果。分割这个命令行：

- ps 告诉我们有关进程的信息。
- -A 列出所有的进程。
- -o pid、size、vsize、cmd 输出 PID、大小信息和命令的名字。
- grep 被用来过滤掉所有其他的结果并且只留下给我们演示的行。

父进程（PID 11268）和它的 4 个派生（fork）子进程显示在了输出中。结果类似于我们在 htop 中所见的。我们可以使用 pmap 来看看每个进程的内存映射，用-x 来请求扩展输出。我们 grep 出模式 s-来列出标记为正在共享的内存块。在父进程和子进程中，我们看见一个 6250000KB（6.2GB）的块在它们之间共享。

例 9-27　使用 pmap 和 ps 来查看从操作系统视角中看到的进程

```
$ ps -A -o pid,size,vsize,cmd | grep np_shared
11268 232464 6564988 python np_shared.py
11288 11232 6343756 python np_shared.py
11289 11228 6343752 python np_shared.py
11290 11228 6343752 python np_shared.py
11291 11228 6343752 python np_shared.py
```

```
ian@ian-Latitude-E6420 $ pmap -x 11268 | grep s-
Address           Kbytes     RSS   Dirty Mode    Mapping
00007f1953663000 6250000 6250000 6250000 rw-s-  zero (deleted)
...
ian@ian-Latitude-E6420 $ pmap -x 11288 | grep s-
Address           Kbytes     RSS   Dirty Mode    Mapping
00007f1953663000 6250000 1562512 1562512 rw-s-  zero (deleted)
```

例9-28显示了为共享这个数组所采取的重要步骤。我们使用一个multiprocessing.Array 来分配一块共享内存作为一个 1 维数组。接着我们从这个对象中实例化了一个 numpy 数组，并把它重塑回一个 2 维数组。现在我们有一个 numpy 包装的内存块，能够在进程间共享，并且能够像一个普通 numpy 数组那样来寻址。numpy 没有管理 RAM，multiprocessing.Array 在管理它。

例 9-28　使用 multiprocessing 来共享 numpy 数组

```
import os
import multiprocessing
from collections import Counter
import ctypes
import numpy as np
from prettytable import PrettyTable

SIZE_A, SIZE_B = 10000, 80000  # 6.2GB - starts to use swap (maximal RAM usage)
```

在例 9-29 中,我们可以看见每个派生(fork)进程访问了一个全局的main_nparray。派生（fork）进程有一个 numpy 对象的拷贝，对象所访问的底层字节作为共享内存来存储。我们的 worker_fn 将使用当前的进程标识符来覆写一个被选取的行（通过 idx）。

例 9-29　worker_fn 为共享 numpy 数组使用 multiprocessing

```
def worker_fn(idx):
    """Do some work on the shared np array on row idx"""
    # confirm that no other process has modified this value already
    assert main_nparray[idx, 0] == DEFAULT_VALUE
    # inside the subprocess print the PID and id of the array
    # to check we don't have a copy
    if idx % 1000 == 0:
           print " {}: with idx {}\n id of local_nparray_in_process is {} in PID {}"\
            .format(worker_fn.__name__, idx, id(main_nparray), os.getpid())
# we can do any work on the array; here we set every item in this row to
# have the value of the process ID for this process
main_nparray[idx, :] = os.getpid()
```

在我们例 9-30 的__main__中，我们通过三个主要阶段来工作：

1．构建一个共享的 multiprocessing.Array 并却把它转换成一个 numpy 数组。

2．给数组设置一个默认值，并生成 4 个进程来并行地在数组上工作。

3．在进程返回后，验证数组的内容。

典型情况下，你设置了一个 numpy 数组，并在一个单独的进程中工作，可能就像 arr = np.array((100, 5), dtype = np.float_)那样来做些事情。这在一个单独进程中不错，但是你不能跨进程来共享数据，既不能写，也不能读。

技巧就是创建一个共享的字节块。一种方式是创建一个 multiprocessing.Array。Array 默认包在锁中来防止并发编辑，但是我们不需要这把锁，因为我们会对我们的共享模式小心翼翼。为了清晰地与其他组员进行沟通，把它显式化并设置 lock = False 是值得的。

如果你不去设置 lock = False，那么你会得到一个对象，而不是一个对字节的引用，并且你需要调用.get_obj()来得到字节。通过调用.get_obj()，你绕开了锁，所以在第一步中，不显式地去做，就没有任何值。

接下来我们就采用这个共享字节块，并使用 frombuffer 来包装成一个 numpy 数组。dtype 是可选的，但是既然我们是在传送字节，显式化总是合理的。我们做了重塑，这样我们就能够以一个 2 维数组来寻址字节。数组值默认设置成了 0。例 9-30 显示出我们的__main__是满的。

例 9-30　为共享而设置 numpy 数组的主函数

```
if __name__ == '__main__':
    DEFAULT_VALUE = 42
    NBR_OF_PROCESSES = 4

    # create a block of bytes, reshape into a local numpy array
    NBR_ITEMS_IN_ARRAY = SIZE_A * SIZE_B
    shared_array_base = multiprocessing.Array(ctypes.c_double,
                                              NBR_ITEMS_IN_ARRAY, lock=False)
    main_nparray = np.frombuffer(shared_array_base, dtype=ctypes.c_double)
    main_nparray = main_nparray.reshape(SIZE_A, SIZE_B)
    # assert no copy was made
    assert main_nparray.base.base is shared_array_base
    print "Created shared array with {:,} nbytes".format(main_nparray.nbytes)
    print "Shared array id is {} in PID {}".format(id(main_nparray), os.getpid())
    print "Starting with an array of 0 values:"
    print main_nparray
    print
```

为了证实我们的进程是在我们所启动的相同的数据块上操作，我们会为每一项设置一个新的 DEFAULT_VALUE——所以你会看到在例 9-31 的顶部（我们用生命、宇

宙和万物的答案)。下一步，我们构建了一个进程池（在这个例子中是 4 个进程），接着通过调用 map 来批量发送行索引。

例 9-31　使用 multiprocessing 来共享 numpy 数组的主函数

```
# modify the data via our local numpy array
main_nparray.fill(DEFAULT_VALUE)
print "Original array filled with value {}:".format(DEFAULT_VALUE)
print main_nparray

raw_input("Press a key to start workers using multiprocessing...")
print

# create a pool of processes that will share the memory block
# of the global numpy array, and share the reference to the underlying
# block of data so we can build a numpy array wrapper in the new processes
pool = multiprocessing.Pool(processes=NBR_OF_PROCESSES)
# perform a map where each row index is passed as a parameter to the
# worker_fn
pool.map(worker_fn, xrange(SIZE_A))
```

一旦我们完成了并行处理，我们回到父进程来验证结果（例 9-32）。验证步骤在数组上通过一个平面化的视图来运行（注意，视图不做拷贝，它只是在 2 维数组上创建了一个 1 维的可迭代视图），为每个 PID 的频率计数。最后，我们执行了一些 assert 检查来确保我们得到了期望的计数。

例 9-32　验证共享结果的主函数

```
print "Verification - extracting unique values from {:,} items\nin the numpy
    array (this might be slow)...".format(NBR_ITEMS_IN_ARRAY)
# main_nparray.flat iterates over the contents of the array, it doesn't
# make a copy
counter = Counter(main_nparray.flat)
print "Unique values in main_nparray:"
tbl = PrettyTable(["PID", "Count"])
for pid, count in counter.items():
    tbl.add_row([pid, count])
print tbl

total_items_set_in_array = sum(counter.values())

# check that we have set every item in the array away from DEFAULT_VALUE
assert DEFAULT_VALUE not in counter.keys()
# check that we have accounted for every item in the array
assert total_items_set_in_array == NBR_ITEMS_IN_ARRAY
# check that we have NBR_OF_PROCESSES of unique keys to confirm that every
# process did some of the work
assert len(counter) == NBR_OF_PROCESSES

raw_input("Press a key to exit...")
```

我们只创建了一个 1 维的字节数组，把它转换为一个 2 维数组，在 4 个进程间共享数组，并允许它们在相同的内存块上并发处理。这种方法有助你在许多核上搞并行化。然而，要小心对相同数据点的并发存取——如果你想要避免同步的问题，你就不得不在 multiprocessing 中使用锁，这会拖慢你的代码。

9.7 同步文件和变量访问

在下面的例子中，我们会看看多进程共享和操控一个状态——在这种情况下，4 个进程以一定次数递增一个共享的计数器。缺少同步过程的话，计数就是不正确的。如果你要以一种一致性的方式来共享数据的话，你总是需要一个方法来同步数据的读写，不然你就会在错误中结束。

典型情况下，同步方法和你所使用的特定操作系统（OS）息息相关，而且它们还常常和你所使用的特定语言息息相关。在这里，我们就看看使用 Python 库的基于文件的同步，在 Python 进程间共享一个整数对象。

9.7.1 文件锁

读写一个文件是在本节中共享数据的最慢的例子。

你可以在例 9-33 中看看我们第一个工作函数。该函数在一个局部计数器上做迭代。在每一次迭代中，它打开了一个文件，读取已存在的值，自增 1，然后用新的值覆写掉老的值。在第一次迭代中，文件会是空的或者不存在，所以它会捕捉一个异常并假设值应该为零。

例 9-33 没有锁的工作函数

```
def work(filename, max_count):
    for n in range(max_count):
        f = open(filename, "r")
        try:
            nbr = int(f.read())
        except ValueError as err:
            print "File is empty, starting to count from 0, error: " + str(err)
            nbr = 0
        f = open(filename, "w")
        f.write(str(nbr + 1) + '\n')
        f.close()
```

让我们用一个进程来运行这个例子。你能在例 9-34 中看见输出。工作函数被调用了 1000 次，正如所期望的那样，它计数正确，没有损失任何数据。在第一次读取

时，见到了一个空文件。这会为 int()抛出 invalid literal for int()的错误（因为在一个空字符串上调用了 int()）。这个错误只发生了一次，之后，我们总是会有一个合法的值用于读取并把它转变成一个整数。

例 9-34　不用锁，用一个进程来做基于文件的计数的用时

```
$ python ex1_nolock.py
Starting 1 process(es) to count to 1000
File is empty, starting to count from 0,
error: invalid literal for int() with base 10: ''
Expecting to see a count of 1000
count.txt contains:
1000
```

现在我们将用 4 个并发进程来运行相同的工作函数。我们没有任何加锁的代码，所以我们将期望有一些奇怪的结果。

问题

在你查看下面的代码前，当两个进程同时从相同的文件读取或写入的时候，你会期待看见哪两种类型的错误呢？思考一下代码的两种主要状态（每个进程的开始执行处和每个进程的正常运行状态）。

瞧例 9-35 来看这个问题。首先，当每个进程启动时，文件是空的，所以它们都设法从零开始计数。第二，当一个进程写时，另一个进程能够读到一个部分写完的不能被解析的结果。这会导致异常，就会写回零。这样依次进行，导致我们的计数器保持在重置状态！你能看到\n 和两个值如何被两个并发进程写入到同样的打开文件中，导致第三个进程读取了一个无效项吗？

例 9-35　不用锁，使用 4 个进程基于文件的计数的用时

```
$ python ex1_nolock.py
Starting 4 process(es) to count to 4000
File is empty, starting to count from 0,
error: invalid literal for int() with base 10: ''
File is empty, starting to count from 0,
error: invalid literal for int() with base 10: '1\n7\n'
# many errors like these
Expecting to see a count of 4000
count.txt contains:
629
$ python -m timeit -s "import ex1_nolock" "ex1_nolock.run_workers()"
10 loops, best of 3: 125 msec per loop
```

例 9-36 展示了用 4 个进程调用工作函数的 multiprocessing 代码。注意我们没

有使用 map，而是构建了一个 Process 对象的列表。尽管我们在这里不使用它的功能性，但 Process 对象给予我们能力来内省每个进程的状态。我们鼓励你读读文档来学习为什么你可能想要使用 Process。

例 9-36 run_workers 设置 4 个进程

```
import multiprocessing
import os

...
MAX_COUNT_PER_PROCESS = 1000
FILENAME = "count.txt"
...

def run_workers():
    NBR_PROCESSES = 4
    total_expected_count = NBR_PROCESSES * MAX_COUNT_PER_PROCESS
    print "Starting {} process(es) to count to {}".format(NBR_PROCESSES,
                                                          total_expected_count)
    # reset counter
    f = open(FILENAME, "w")
    f.close()

    processes = []
    for process_nbr in range(NBR_PROCESSES):
        p = multiprocessing.Process(target=work, args=(FILENAME,
                                                       MAX_COUNT_PER_PROCESS))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

    print "Expecting to see a count of {}".format(total_expected_count)
    print "{} contains:".format(FILENAME)
    os.system('more ' + FILENAME)


if __name__ == "__main__":
    run_workers()
```

使用 lockfile 模块，我们能够引入一种同步方法，这样在同一时刻只有一个进程写，其他进程都要等待轮到它们的时候。因此整体过程运行得更慢，但不会犯错。你可以在例 9-37 中看到正确的输出。你会发现在线的完整文档。注意加锁机制和 Python 息息相关，这样其他正在查看这个文件的进程不会关心这个文件已经“被加锁”的本质。

例 9-37　使用锁和 4 个进程基于文件的计数的用时

```
$ python ex1_lock.py
Starting 4 process(es) to count to 4000
File is empty, starting to count from 0,
error: invalid literal for int() with base 10: ''
Expecting to see a count of 4000
count.txt contains:
4000
$ python -m timeit -s "import ex1_lock" "ex1_lock.run_workers()"
10 loops, best of 3: 401 msec per loop
```

使用 lockfile 只是增加了几行代码。首先，我们创建了一个 FileLock 对象，文件名可以是任意的，但使用和你要加锁的文件相同的名字让命令行调试变得更简单。当你请求得到锁时，FileLock 用相同的名字打开了一个新文件，用 .lock 作为后缀名。

没有任何参数的 acquire 会无限期阻塞，直到锁变得可用。一旦你拿到了锁，你就能够做你的处理，而没有任何冲突风险。接着一旦你写完（例 9-38），你就可以释放锁。

例 9-38　有锁的工作函数

```
def work(filename, max_count):
    lock = lockfile.FileLock(filename)
    for n in range(max_count):
        lock.acquire()
        f = open(filename, "r")
        try:
            nbr = int(f.read())
        except ValueError as err:
            print "File is empty, starting to count from 0, error: " + str(err)
            nbr = 0
        f = open(filename, "w")
        f.write(str(nbr + 1) + '\n')
        f.close()
        lock.release()
```

你可以使用一个上下文管理器，在这种情况下，你用 lock：来代替 acquire 和 release。这给运行时增加了少量的开销，但是也让代码变得更容易读一点。清晰度常常优于执行速度。

你也能够用一个 timeout 来请求 acquire 锁，检查已经存在的锁，并打断已经存在的锁。提供了几种加锁机制，对每个平台敏感的默认选项隐藏在了 Filelock 接口后面。

9.7.2 给 Value 加锁

multiprocessing 模块在进程间提供了几个选项来共享 Python 对象。我们能使用低开销的通信来共享基础对象，也能用一个 Manager 来共享更高级别的 Python 对象（例如，字典和列表）（但要注意同步开销会显著地减慢数据共享）。

在这里，我们将使用一个 multiprocessing.Value 对象在进程间共享一个整数。尽管 Value 有锁，但是锁没有尽如你意——它阻止了同时读取或写入，但是没有提供一个原子的递增。例 9-39 演示了这种情况。你能看到我们以一个不正确的计数来终结，这就类似于我们在之前看到的基于文件的不同步的例子。

例 9-39 无锁导致计数不正确

```
$ python ex2_nolock.py
Expecting to see a count of 4000
We have counted to 2340
$ python -m timeit -s "import ex2_nolock" "ex2_nolock.run_workers()"
100 loops, best of 3: 12.6 msec per loop
```

数据没有发生损坏，但是我们错失了好几次更新。如果你从一个进程写入一个 Value，再在另外的进程中消费那个 Value（但不修改），这种方式就可能是合适的。

共享 Value 的代码显示在了例 9-40 中。我们不得不声明一个数据类型和一个初始值——使用 Value("i", 0)，我们请求一个初始值为 0 的有符号整数。它被当作一个常规参数传递给我们的 Process 对象，Process 对象负责在后台进程间共享相同的字节块。为了访问由我们的 Value 所持有的基础对象，我们使用了.value。注意我们正请求一个原地的加法——我们期待它变为一个原子操作，但是 Value 却不支持，所以我们最终的计数比预期要低。

例 9-40 没有锁的计数代码

```
import multiprocessing

def work(value, max_count):
    for n in range(max_count):
        value.value += 1

def run_workers():
...
    value = multiprocessing.Value('i', 0)
    for process_nbr in range(NBR_PROCESSES):
        p = multiprocessing.Process(target=work, args=(value, MAX_COUNT_PER_PROCESS))
        p.start()
        processes.append(p)
...
```

我们能够增加一个 Lock，它就会以与我们之前所看到的 FileLock 例子很类似的方式来工作。你能在例 9-41 中看到正确同步后的计数。

例 9-41　使用 Lock 来同步写一个 Value

```
# lock on the update, but this isn't atomic
$ python ex2_lock.py
Expecting to see a count of 4000
We have counted to 4000
$ python -m timeit -s "import ex2_lock" "ex2_lock.run_workers()"
10 loops, best of 3: 22.2 msec per loop
```

在例 9-42 中，我们已经使用了一个上下文管理器（有 Lock）来获取锁。就如在之前的 FileLock 例子中那样，它无限等待来获取锁。

例 9-42　使用 context manager 来获取锁

```
import multiprocessing


def work(value, max_count, lock):
    for n in range(max_count):
        with lock:
            value.value += 1


def run_workers():
...
    processes = []
    lock = multiprocessing.Lock()
    value = multiprocessing.Value('i', 0)
    for process_nbr in range(NBR_PROCESSES):
        p = multiprocessing.Process(target=work,
                                    args=(value, MAX_COUNT_PER_PROCESS, lock))
        p.start()
        processes.append(p)
...
```

就如在 FileLock 例子中的那样，避免使用上下文管理器会快一点。例 9-43 中的片段显示了怎样使用和释放 Lock 对象。

例 9-43　内联加锁，而不用上下文管理器

```
lock.acquire()
value.value += 1
lock.release()
```

既然 Lock 没有给予我们所追求的细粒度，它提供的基础锁浪费了一点不必要的时间。我们能够如例 9-44 中的那样用一个 RawValue 取代 Value，并取得一个递增的速度提升。如果你有兴趣看看在这个变化背后的字节码，那么就读一下 Eli Bendersky 关于这个主题的博客帖子。

例 9-44　展示最快的 RawValue 和 Lock 方法的控制台输出

```
# RawValue has no lock on it
$ python ex2_lock_rawvalue.py
Expecting to see a count of 4000
We have counted to 4000
$ python -m timeit -s "import ex2_lock_rawvalue" "ex2_lock_rawvalue.run_workers()"
100 loops, best of 3: 12.6 msec per loop
```

为了使用 `RawValue`，只要如例 9-45 中所示的那样和 `Value` 交换就可以了。

例 9-45　使用 RawValue 整数的例子

```
...
def run_workers():
...
    lock = multiprocessing.Lock()
    value = multiprocessing.RawValue('i', 0)
    for process_nbr in range(NBR_PROCESSES):
        p = multiprocessing.Process(target=work,
                                    args=(value, MAX_COUNT_PER_PROCESS, lock))
        p.start()
        processes.append(p)
```

如果我们要共享一个基础对象数组，我们也可以使用 `RawArray` 来代替一个 `multiprocessing.Array`。

随着在多进程间共享一个标记和同步数据共享，我们已经看到了各种不同的方式来在一台单独的机器上的多进程间划分工作。然而，请记住数据共享能够产生令人头疼的问题——设法尽可能的避开它。让一台机器处理共享状态的所有边边角角的情况是困难的，当你第一次被迫调试多进程交互时，你就会意识到为什么为人所接受的智慧就是尽量避免这种情况。

确实要考虑写出运行慢一点但是更容易被你的团队所理解的代码。使用一个类似 Redis 的外部工具来共享状态会生成一个在运行时能够被非开发者所检查的系统——这是一种强大的方式来让你的团队监控在你的并行系统中所发生的事情。

一定要记住调试过性能的 Python 代码更不可能被你团队中的更初级的员工所理解——他们或害怕它，或会破坏它。避免这个问题（接受在速度上的牺牲）来保持团队的高效率。

9.8　小结

在本章中我们已经涉及了很多。首先我们看了两个令人窘迫的并行问题，其中一个具有可预料的复杂性，而另一个具有不可预料的复杂性。当我们在第 10 章讨论集

群时，我们会再次短暂地在多台机器上使用这些例子。

接下来，我们看到在 multiprocessing 中对 Queue 的支持和它的开销。一般情况下，我们推荐使用一个外部的队列库，这样队列的状态更加透明。你应该倾向于使用一个容易阅读的工作格式而不是序列化（pickled）的数据，这样就容易调试。

进程间通信（IPC）的讨论应该让你对有效使用 IPC 的难度印象深刻，仅仅使用一个天真的并行方式（没有 IPC）可能是有意义的。购买一台具有更多核的更快的计算机可能比设法使用 IPC 来开发一台现有的机器要现实得多。

不做拷贝的并行共享 numpy 矩阵仅仅对于一小撮问题是重要的，但是当它重要时，它就真的重要。确保你真的没有在进程间拷贝数据需要花费额外的几行代码和一些安全检查。

最后，我们看了使用文件和内存锁来避免损坏数据——这是细微和难以跟踪的错误的来源，本节向你展示了一些鲁棒和轻量级的解决方案。

在下一章中我们会看看使用 Python 的集群。使用集群，我们可以超越单机的并行性并利用一组机器的 CPU。这引入了一个调试痛苦的新世界——不仅仅是你的代码可能有错，而且其他机器也可能有错误（或是错误配置，或是硬件失效）。我们会展示如何来使用并行的 Python 模块并行化 pi 的估算演示，并展示如何使用一个 IPython 集群来运行 IPython 内部的研究代码。

第 10 章

集群和工作队列

读完本章之后你将能够回答下列问题

- 为什么集群是有用的？
- 集群的代价是什么？
- 我该如何把一个多进程的解决方案转换成一个集群解决方案？
- IPython 集群如何工作？
- NSQ 是怎样有助于创建鲁棒的生产系统？

一个集群通常被视作一组共同工作来解决公共问题的计算机集合。从外部看来，它可能就是一个更大的独立系统。

在 20 世纪 90 年代，在一个本地局域网上使用一组商业 PC 的集群来进行集群化处理的概念变得流行起来——被称作 Beowulf 集群。后来 Google 在自己的数据中心通过使用商业 PC 集群的实践，得到了一个巨大的提升，尤其是针对运行 MapReduce 的任务。在天平的另一端，TOP500 项目每年对最强大的计算机系统进行排名，这些系统都具有典型的集群化的设计，并且最快的机器都使用了 Linux。

亚马逊 Web 服务（AWS）通常既被用来做云中的工程产品集群，又被用来为短期的项目比如机器学习来按需构建集群。使用 AWS，你能够租用多台八核 Intel Xeon，60GB 的 RAM 的机器，每小时单台 1.68 美元，还有 244GB RAM 的机器以及具有 GPU 的机器。如果你想要为计算密集型的任务探索 AWS 的临时集群，可以看看第 10.6.2 节和 StarCluster 包。

不同的计算任务需要不同配置、不同大小、不同容量的集群。在本章中我们会定义一些公共的场景。

在你转移到集群化的解决方案时，确保你已经：

- 度量了你的系统，因此你理解瓶颈在哪里。
- 探索了类似 Cython 的编译器解决方案。
- 在一台单独的机器上利用了多核（可能一台大型机器上有很多核）。
- 探索了使用更少 RAM 的技术。

让你的系统保持运行在一台机器上（即使“一台机器”其实是强大的具有许多 RAM 和 CPU 的计算机）将会让你的生活更轻松些。如果你真的需要许多 CPU 或者并行处理来自磁盘上的数据的能力，抑或是具有类似高弹性和高响应速度之类的产品需求，那么请转移到一个集群上去。

10.1　集群的益处

集群最明显的益处就是你能够轻易地扩展计算需求——如果你需要处理更多的数据或者得到更快的答案，你只要增加更多的机器（或“节点”）。

通过增加机器，你也能提高可靠性。每个机器组件有一定的失效概率，而设计良好的话，一定数量的组件失效就不会让整个集群停止工作。

集群也被用来创建动态扩展的系统。一种常见的使用场景就是集群化一组服务器来处理 web 请求或相关联的数据（例如，缩放用户照片、视频转码，或是语音转录），并且在一天中的某些时段里请求增加时，就激活更多的服务器。

只要机器激活时间足够快从而赶上处理需求变化的速度，动态扩容就是处理非均匀的应用模式的一种非常节约成本的方式。

集群化的更细微的收益就是集群能够按地理来分割，但还是受到中心化的控制。如果一个地理区域遭受断电（例如，洪水或电力损失），其他的集群还能继续工作，也许更多的处理单元被添加进来处理请求。集群也允许你运行在异构的软件环境上（例如，不同版本的操作系统和处理软件），这或许能够提高整体系统的鲁棒性——然而要注意那一定是一个专家级别的主题！

10.2 集群的缺陷

转移到集群化的解决方案需要转变思想。从串行转到我们在第 9 章中所介绍过的并行代码，是一个需要转变思想的演进。突然间，你不得不考虑当你有超过一台机器时，会发生什么——在机器间会有延迟，你需要知道你的其他机器是否在工作，你还需要让所有机器运行相同版本的软件。系统管理可能是你最大的挑战。

此外，你通常不得不努力思考你要实现的算法以及一旦你拥有了所有这些可能需要保持同步的额外的转移部分，将会发生什么事。这个额外计划可能施加了一个沉重的智商税，可能让你从核心任务中分心走岔，一旦系统成长得足够庞大，你可能需要一个专职的工程师来加入你的团队。

备忘

我们尝试集中于有效使用一台机器的理由就是因为我们都相信如果你只处理一台计算机而不是一组计算机，那么生活就会更轻松（尽管我们承认玩弄一个集群会更有趣——直到它失效为止）。如果你能够垂直扩展（通过购买更多的 RAM 或者 CPU），那么就值得调查这种方法对集群的支持。当然，你的处理需求可能会超过垂直扩展所能做的一切，或者一个集群的鲁棒性可能比一台单独的机器更加重要。然而，如果你是独自一个人工作于这个任务，也要记住运行一个集群会消耗你的一些时间。

当设计一个集群化的解决方案时，你需要记住每台机器的配置可能是不相同的（每台机器会有不同的负载和不同的局部数据）。你该如何来把所有正确的数据放到处理你任务的机器上来呢？移动任务和数据涉及的延迟会成为问题吗？你的任务需要和其他任务相互通信部分结果吗？当几个任务正在运行时，如果一个进程失效了或者一台机器挂了或者一些硬件擦除了自己，那么会发生什么呢？如果你不去考虑这些问题，失败就会随之而来。

你也应该考虑到失效是能够被接受的。例如，当你运行一个基于内容的 Web 服务时，你可能不需要 99.999%的可靠性——如果一项任务偶然失效了（例如，一张图片没有足够快速地被缩放），并且需要用户来重载页面，那是每个人都已经习以为常的。那可能不是你想要给予用户的解决方案，但是接受一点失效常常降低了你的边际工程和管理成本，那是值得的。另一方面，如果一个高频交易系统经历了失效，那么为糟糕的股票市场交易所付出的代价可能是相当巨大的！

维护一个固定的基础设施可能变得代价高昂。采购机器是相对廉价的，但是它们具有一个糟糕的变坏的趋势——自动软件更新会有小故障，网卡会失效，硬盘会有写错误，电力供给可能输出损坏数据的尖峰能量，宇宙射线可能反转 RAM 模块的一个比特位。你拥有越多的计算机，就会损失越多的时间来处理这些问题。你迟早会想要增加一名系统工程师来处理这些问题，因此又增加了 100000 美元的预算。使用一个基于云的集群能够缓解许多这些问题（它花费更多钱，但是你不必处理硬件维护），而且一些云供应商也为廉价而临时的计算资源提供了一个即时付费的市场。

伴随着集群随时间有机成长而带来的潜在问题就是如果一切都关掉了，可能没有人对怎样安全地重启集群而制定文档。如果你没有一个文档化的重启计划，那你就应该设想你将不得不在可能是最坏的时候来写文档（你的其中一个作者已经卷入到在圣诞夜调试这种类型的问题——这不是你想要的圣诞礼物！）。在这点上，你也会了解到让一个系统的每个组件开始加速可能要花费多久——也许集群的每个组件要花费几分钟来启动并开始处理任务，所以如果你有 10 个依次运行的组件，可能要花费一小时来让整个系统完成冷启动。结果就是你可能有一个小时之久的堆积数据。那么你有所需的容量来及时处理这些堆积数据吗？

懈怠的行为可能是引起代价高昂的错误的原因，而复杂且难以预料的行为可能导致代价高昂的不可预料的结果。让我们看看两起引人注目的集群失效，并看看我们能够学到的教训。

10.2.1 糟糕的集群升级策略造成华尔街损失 4.62 亿美元

在 2012 年，高频交易公司骑士资本在集群中做软件升级期间引入了一个错误，损失了 4.62 亿美元。软件做出了超出客户所请求的股票买卖。

在交易软件中，一个更老的标记转用于新函数。升级已进行到了 8 台活跃机器中的 7 台，但是第 8 台机器使用了更旧的代码来处理标记，这导致了所做出的错误的交易。安全和交易委员会（SEC）注意到骑士资本没有让其他技术人员来检查升级，并且没有流程来检查已经存在的升级。

根本的错误看起来有两个原因。首先，软件开发过程没有移除一个废弃的功能，所以僵尸代码遗留了下来。第二就是没有人工检查过程来确认升级成功完成了。

技术债最终增添了不得不付出的代价——倾向于在没有压力时花时间来解决技术债。在构建和重构代码时，总是使用单元测试。缺乏一个手写的检查列表在系统升级期间去核对一遍，又缺乏第二双眼睛来检查，可能就会让你付出代价高昂的失败。

飞机驾驶员有理由必须要过一遍下降检查列表：这意味着没有人会错过重要的步骤，无论他们以前可能已经做了多少次。

10.2.2 Skype 的 24 小时全球中断

Skype 在 2010 年遭受了一次 24 小时全球范围的失效。在幕后，Skype 由点对点网络所支持。部分系统发生的过载（用来处理线下的即时消息）造成了 Windows 客户端的响应延迟，一些版本的 Windows 客户端没有合适地处理延迟的响应就垮掉了。总体上，大约 40%的活跃客户端垮掉了，包括 25%的公共超级节点。超级节点对网络上的路由数据起关键作用。

随着 25%的路由断线（它恢复了，但是缓慢地恢复），整体网络处于严重的压力下。崩溃的 Windows 客户节点也正在重启中，并且尝试重新加入网络，在已经过载的系统上增加了新的大量的流量。超级节点在经受太多的负载时，有回退步骤，所以它们开始关闭来对流量的波浪做出反应。

Skype 变得 24 小时严重不可用。恢复步骤涉及首先要设置几百台新的万兆超级节点，它们被配置成用以处理增加的流量，接着用几千台更多的新超级节点来跟进。在接下来的几天内，网络恢复了。

事故造成了 Skype 的很多窘境，显然，在几天紧张的日子里，他们的精力也切换到限制破坏中去了。客户被迫寻找语音电话的可替代方案，可能对竞争者来说是一个市场恩赐。

考虑到网络的复杂性以及发生失效的升级，这个失效可能是难以预测和做出计划来应对的。网络上的所有节点不会失效的理由是因为不同的软件版本和不同的平台——异构网络比同构网络更具有可靠性的收益。

10.3 通用的集群设计

通常由合理等价的机器所组成的一个局部的临时集群来开始。你可能想知道你是否能给一个临时网络增加旧机器，但是更老的 CPU 常常消耗很多电力而且运行得非常慢，所以相比一个新的高规格的机器，它们无法如你所期望的那样做出贡献。一个在公司的集群需要有能够维护的人。一个连接亚马逊 EC2 或者微软 Azure 或者一个学术机构的集群解除了对供应者团队的硬件支持。

如果你有理解良好的处理需求，设计一个定制化的集群可能是有意义的——也许是一个使用无限带宽的高速互连取代千兆以太网，或者是一个使用特定配置的 RAID

驱动来支持你的读、写或弹性化需求。你可能想要在一些机器上混合 CPU 和 GPU，或只是默认为 CPU。

你可能想要一个大规模去中心化的处理集群，就像由类似于通过伯克利网络计算系统开放基础设施（BOINC）所开发的 SETI@home 和 Folding@home 之类的项目所使用的那样——它们共享了中心化的协调系统，但是计算节点以即时的方式加入和离开项目。

在硬件设计之上，你能够运行不同的软件架构，工作队列最通用而且最容易理解。任务常常被放入一个队列并由一个处理者所消费。处理结果可能进入另一个队列来做进一步处理，或者用来作为最终的结果（例如，被添加进一个数据库）。消息传递系统稍有不同——消息被放入一个消息总线，接着由其他机器所消费。消息可能超时而得以删除，并且可能由多台机器所消费。一个更复杂的系统就是当进程使用进程间通信与其他进程交流的时候——这可以被考虑成一个专家级别的配置，因为有许多险途来把它搞坏，导致你丧失了理智。只有当你真正知道需要它时，才走上进程间通信（IPC）的道路。

10.4 怎样启动一个集群化的解决方案

启动一个集群化系统的最简单的方式就是从一台既运行作业服务器又运行作业处理器（和 CPU 是一对一的关系）的机器开始。如果你的任务是 CPU 密集型的，每一个 CPU 跑一个作业处理器；如果你的任务是 I/O 密集型的，每一个 CPU 跑几个作业处理器。如果你的任务是 RAM 密集型的，请小心不要耗尽 RAM。让你的单机解决方案在一个处理器上工作正常，接着再增加更多。让你的代码以不可预料的方式失效（例如，在你代码中做 `1/0`，对你的工作者使用 `kill -9 <pid>`，从插座上拔掉电源，这样让整个机器挂掉）来检查你的系统是否健壮。

显然，你想要做比这更重量级的测试—— 一个充满了编码错误和人工异常的单元测试集就好。Ian 喜欢抛出非预期的事件，就像让一个处理器运行一个作业集，而一个外部进程正系统化地杀掉重要的进程并且通过任何你所使用的监控进程来确认所有这些进程都干净地重启了。

一旦你有了一个运行的作业处理器，就增加第二个。要检查你没有使用太多的 RAM。你是否以从前两倍的速度来处理任务？

现在引入了第二台机器，仅仅只有一个作业处理器跑在新机器上，在协作机器上没有作业处理器。它处理作业的速度是否与协作机器有处理器时一样快？如果不是，

为什么？是延迟的问题吗？你做了不同的配置吗？也许你有不同的机器硬件，类似于 CPUs、RAM 和缓存大小？

现在加入另外 9 台计算机来测试看看你是否以比从前快 10 倍的速度来处理任务。如果不是，那么为什么呢？是否发生了网络冲突减慢了你的整体处理速率？

当机器启动时，为了可靠地启动集群组件，我们倾向于使用 `cron` 任务，`Circus` 或 `supervisord`，或者有时使用 `Upstart`（正在被 `systemd` 所代替）。`Circus` 比 `supervisord` 更新，但两者都是基于 Python 的。`cron` 陈旧，但是如果你只是启动一个类似于能启动所需子进程的监控进程那样的脚本的话，它是非常可靠的。

一旦你有了一个可靠的集群，你可能就想要引入一个类似 Netflix 的 ChaosMonkey 那样的随机杀手工具来故意杀掉你的部分系统来测试它们的弹性。你的进程和硬件最终会挂掉，但你不需要在经历痛苦之后才知道，你至少可以在遭受你所预料会发生的错误中存活下来。

10.5 使用集群时避免痛苦的方法

Ian 遭受过的一个特别痛苦的经历是一个集群化系统中的一系列查询陷入了停顿状态。最近的查询没有被消费，所以它们被堆积起来了。一些机器跑完了内存，所以它们的进程挂掉了。之前的查询正在被处理，但是没有把它们的结果传递给下一个队列，所以它们崩溃了。最后，第一个队列填充满了，但没有被消费，所以它崩溃掉了。然后我们为最终丢失了来自提供者的数据而付出了代价。你必须拟定出一些注意事项来考虑你的集群以各种各样的方式挂掉（不是如果它挂掉，而是当它挂掉的时候）以及会发生什么结果。你会丢失数据吗（这是个问题吗？）？你会有一个巨大的难以处理的积压任务吗？

有一个容易调试的系统可能胜过有一个更快的系统。工程时间以及失效时间的代价可能是你最大的开销（如果你正运行一个导弹防御程序，这就不恰当，但是对于一个创业公司来说是恰当的）。当传递消息时，与其使用一个低级的压缩的二进制协议来削减一些字节，不如使用人类可读的 JSON 文本。传递和解码消息的确增加了开销，但是当你剩下一个特别的数据库时，当一台核心计算机着火后，你就会庆幸当你努力把系统恢复上线时，你能够迅速读取出重要的信息。

确保花费少量时间和廉价的金钱来给系统部署升级——无论是操作系统升级还是你的软件新版本。每当集群中有任何改变时，如果它处于一个反复无常的状态，系统就会以一种古怪的方式来响应，你就会冒风险。确保你使用一个部署系统，类似

于 Fabric、Salt、Chef 或 Puppet，或者一个类似于 Debian 的.deb，RedHat 的 rmp 或者类似于一个亚马逊机器镜像。能够健壮地部署一个升级整个集群的更新（在任何发现的问题上有报道）在困难期间大大减轻了压力。

正面报告是有用的。每天发一封邮件给某些人详述集群的性能。如果邮件没有找到，那就是一个有用的线索来告知发生了一些事情。你可能也想要其他的早期告警系统来更快地通知你，在这里，Pingdom 和 ServerDensity 尤其有用。一个响应缺失事件的“死人开关”是另一个有用的备份（例如，死人的开关）。

把集群的健康状况报告给团队是很有用的。这可能是一个在 web 应用程序内部的管理页面，或者是一个独立的报告。Ganglia 在这方面很给力。Ian 看到了一个在办公室中的运行于一台多余 PC 上的类似于星际迷航中的 LCARS 的接口，当检测到问题时播出“红色警报”的声音——这对引起整个办公室的注意特别有效。我们已经看到了类似于老式风格的锅炉压力测量计的 Arduinos 驱动模拟设备（当指针移动时，发出美妙的声音！）显示出系统的负载。这种类型的报告是重要的，这样每个人就理解了“正常”和“这可能会破坏我们周五晚上的生活”之间的区别。

10.6 三个集群化解决方案

在下面几节中，我们介绍 Parallel Python、IPython Parallel 和 NSQ。

Parallel Python 有一个很熟悉 `multiprocessing` 的接口。把你的 multiprocessing 解决方案从一台单独的多核机器升级到多机器的设置就是个几分钟内的活。Parallel Python 几乎没有依赖性，容易在一个本地集群中为研究工作做配置。它不是很强大还缺少通信机制，但是对于发送令人为难的并行任务来给一个小规模的局部集群来说，是很容易使用的。

IPython 集群是很容易在一台多核机器上使用的。既然许多研究者使用 IPython 作为它们的 shell，使用它来做并行任务控制也是很自然的。构建一个集群需要一点系统管理知识，而且有一些依赖性（例如 ZeroMQ），所以设置起来比 Parallel Python 稍微复杂一点。IPython Parallel 的一个巨大胜利就是你能够如本地集群一样来使用远端集群这样一个事实（例如，使用亚马逊 AWS 和 EC2）。

NSQ 是一个可随时投入生产的队列系统，在类似 Bitly 那样的公司中所使用。它有持久性（所以如果机器挂了，任务就能够被其他的机器重新捡起）和强大的可扩展机制。它具有更强大的能力，对系统管理和工程技巧的需求更大一些。

10.6.1　为简单的本地集群使用 Parallel Python 模块

Parallel Python（pp）模块让本地的工作者集群能够使用一个类似于 multiprocessing 的接口。显而易见，那意味着把代码从使用 map 的 multiprocessing 转变为 Parallel Python 是很方便的。你能够轻易地使用一台机器或一个临时网络来运行代码。你能够使用 pip install pp 来安装它。

我们能够通过 Parallel Python 来使用蒙特卡罗方法，就如我们在 9.3 节中使用本地机器所做的那样——注意在例 10-1 中，这个接口与更早 multiprocessing 的例子是多么相似啊。我们在 nbr_trials_per_process 中创建了一个工作列表，并且把这些任务传递给 4 个本地进程。我们能够创建所需数量的工作项，当工作者变空闲时，它们会被消费。

例 10-1　Parallel Python 的本地例子

```
...
import pp

NBR_ESTIMATES = 1e8

def calculate_pi(nbr_estimates):
    steps = xrange(int(nbr_estimates))
    nbr_trials_in_unit_circle = 0
    for step in steps:
        x = random.uniform(0, 1)
        y = random.uniform(0, 1)
        is_in_unit_circle = x * x + y * y <= 1.0
        nbr_trials_in_unit_circle += is_in_unit_circle
    return nbr_trials_in_unit_circle

if __name__ == "__main__":
    NBR_PROCESSES = 4
    job_server = pp.Server(ncpus=NBR_PROCESSES)
    print "Starting pp with", job_server.get_ncpus(), "workers"
    nbr_trials_per_process = [NBR_ESTIMATES] * NBR_PROCESSES
    jobs = []
    for input_args in nbr_trials_per_process:
        job = job_server.submit(calculate_pi, (input_args,), (), ("random",))
        jobs.append(job)
    # each job blocks until the result is ready
    nbr_in_unit_circles = [job() for job in jobs]
    print "Amount of work:", sum(nbr_trials_per_process)
    print sum(nbr_in_unit_circles) * 4 / NBR_ESTIMATES / NBR_PROCESSES
```

在例 10-2 中，我们拓展了例子——这次我们需要 1024 个任务做 100000000 次估算，每一次用动态配置的集群。在远端机器上，我们能够运行 python ppserver.py

-w 4 -a -d，远端服务器将用 4 个处理器（在 Ian 的笔记本电脑上默认会是 8 个，但是我们不想使用 4 个超线程，所以我们选择了 4 个 CPU），使用自动连接和调试日志。调试日志在屏幕上打印调试信息，对于检查工作是否已经接收到来说，这是有用的。自动连接标记意味着我们不必声明 IP 地址，我们让 pp 自己广播并连接到服务器上。

例 10-2　在集群上的 Parallel Python

```
...
NBR_JOBS = 1024
NBR_LOCAL_CPUS = 4
ppservers = ("*",) # set IP list to be autodiscovered
job_server = pp.Server(ppservers=ppservers, ncpus=NBR_LOCAL_CPUS)

print "Starting pp with", job_server.get_ncpus(), "local workers"
nbr_trials_per_process = [NBR_ESTIMATES] * NBR_JOBS
jobs = []
for input_args in nbr_trials_per_process:
    job = job_server.submit(calculate_pi, (input_args,), (), ("random",))
    jobs.append(job)
...
```

使用第二台强力的笔记本电脑来运行，计算时间大概减半了。另一方面，一台具有单 CPU 的老式 MacBook 几乎没有帮助——它通常以如此慢的速度计算其中一项任务，导致快速的笔记本电脑空闲下来，没有更多的工作来运行，所以整体完成时间比只使用一台快速的笔记本电脑更长久。

这是一个很有用的方法来开始为轻量级的计算任务构建一个临时的集群。你可能不想要在生产环境中使用它（Celery 或者 GearMan 可能是一个更好的选择），但是对于研究目的和易扩展性来说，当知悉一个相关的问题时，它就会让你快速取胜。

pp 无法帮助来分发代码或静态数据给远端的机器，你不得不移动外部库（例如，你可能已经编译成一个静态库的任何东西）到远端机器，并且提供任何的共享数据。它能够序列化（pickle）要运行的代码，处理额外的导入以及你从控制进程所提供的数据。

10.6.2　使用 IPython Parallel 来支持研究

对 IPython 集群的支持通过 ipcluster 到来了。IPython 成为了一个本地和远程处理引擎的接口，数据能够在引擎之间被推送，任务能够被推送到远端机器上。远程调试是有可能的，对消息传递接口（MPI）的支持是可选的。相同的通信机制让 IPython Notebook 接口变得强大。

这对研究设置来说意义巨大——你能够把任务推送到一个在本地集群中的机器，做交互，如果有问题就调试，把数据推送到机器中，并把结果收集回来，所有这一切都以交互的方式来进行。也要注意 PyPy 运行 IPython 和 IPython Parallel。这个组合可能是非常强大的（如果你不使用 numpy）。

在幕后，ZeroMQ 被用来作为消息中间件，所以你需要安装它。如果你在局域网中构建集群，你可以避免使用 SSH 身份验证。如果你需要一定的安全性，那么它完全支持 SSH，但却让配置变得更加复杂一点——从一个可信的局域网上开始，接着当你知悉每个组件如何工作时再扩建。

项目被拆分成 4 个组件。引擎是一个运行代码的同步 Python 解释器。你会运行一组引擎来开启并行计算。控制器提供了引擎的接口，它负责任务分发，并提供了一个直接接口和一个负载均衡接口来提供任务调度。一个中心枢纽用来跟踪引擎、调度器和客户端。调度器隐藏了引擎的同步本质，提供了一个异步接口。

在笔记本电脑上，我们用 ipcluster start -n 4 来启动 4 个引擎。在例 10-3 中，我们启动 IPython 并检查一个本地 Client 能否看到我们的 4 个本地引擎。我们能够使用 c[:]来寻址所有 4 个引擎，我们把一个函数应用于每一个引擎——apply_sync 采用了一个可调用函数，这样我们就提供了一个返回字符串的零参数 lambda 表达式。4 个引擎中的每一个会运行其中一个函数，并返回相同的结果。

例 10-3 测试我们是否能用 IPython 看到本地引擎

```
In [1]: from IPython.parallel import Client

In [2]: c = Client()

In [3]: print c.ids
[0, 1, 2, 3]

In [4]: c[:].apply_sync(lambda:"Hello High Performance Pythonistas!")
Out[4]:
['Hello High Performance Pythonistas!',
 'Hello High Performance Pythonistas!',
 'Hello High Performance Pythonistas!',
 'Hello High Performance Pythonistas!']
```

创建我们的引擎之后，现在它们处于空白状态。如果我们在本地导入模块，它们不会被导入远程引擎。一个既在本地导入又在远端导入的干净的办法就是使用 sync_import 上下文管理器。在例 10-4 中，我们将在本地 IPython 和 4 个连接的引擎上导入 os，接着在 4 个引擎上再次调用 apply_sync 来获取它们的 PIDs。如果我们不做远程导入，我们会得到一个 NameError，因为远程引擎不知道 os 模块。我们也能使用 execue 来在引擎上远程运行任何的 Python 命令。

例 10-4　把模块导入远程引擎

```
In [5]: dview=c[:] # this is a direct view (not a load-balanced view)

In [6]: with dview.sync_imports():
   ....:     import os
   ....:
importing os on engine(s)

In [7]: dview.apply_sync(lambda:os.getpid())
Out[7]: [15079, 15080, 15081, 15089]

In [8]: dview.execute("import sys") # another way to execute commands remotely
```

你会想要把数据推送到引擎。在例 10-5 中显示的 push 命令让你发送字典项来加入每个引擎的全局名字空间。有相应的 pull 来获取这些项：你指定键，它就会从每个引擎返回对应的值。

例 10-5　把共享数据推送到引擎

```
In [9]: dview.push({'shared_data':[50, 100]})
Out[9]: <AsyncResult: _push>

In [10]: dview.apply_sync(lambda:len(shared_data))
Out[10]: [2, 2, 2, 2]
```

现在让我们给集群增加第二台机器。首先，我们将杀死之前所创建的 ipengine 引擎并结束掉 IPython。我们会从一个干净的状态开始。你将需要第二台可用的机器，配置有 SSH 来允许你自动登入。

在例 10-6 中，我们会为集群创建一个新画像。一组配置文件被放入<HOME>/.ipthon/profile_mycluster 目录下。引擎默认被配置成只接受来自 localhost 的连接，而不接受来自外部设备的连接。编辑 ipengine_config.py 来配置 HubFactory，以便接受外部的连接，保存起来，接着使用新画像来启动 ipcluster。我们会回到 4 个本地引擎。

例 10-6　创建一个接受公共连接的本地画像

```
$ ipython profile create mycluster --parallel
$ gvim /home/ian/.ipython/profile_mycluster/ipengine_config.py
# add "c.HubFactory.ip = '*'" near the top
$ ipcluster start -n 4 --profile=mycluster
```

接下来我们需要把这个配置文件传递给我们的远程机器。在例 10-7 中，我们使用 scp 来把 ipcontroller-engine.json（在我们启动 ipcluster 的时候所创建）拷贝到远程机器的.config/ipython/profile_default/security 目录下。一旦拷贝完成，就在远程机器上运行 ipengine。它会在默认目录下查找 ipcontroller-engine.json，如果成功连接了，接着你就会看到类似在这里所显示的消息。

例 10-7　把编辑好的画像拷贝到远程机器并做测试

```
# On the local machine
$ scp /home/ian/.ipython/profile_mycluster/security/ipcontroller-engine.json
      ian@192.168.0.16:/home/ian/.config/ipython/profile_default/security/

# Now on the remote machine
ian@ubuntu:~$ ipengine
...Using existing profile dir: u'/home/ian/.config/ipython/profile_default'
...Loading url_file u'/home/ian/.config/ipython/profile_default/security/
      ipcontroller-engine.json'
...Registering with controller at tcp://192.168.0.128:35963
...Starting to monitor the heartbeat signal from the hub every 3010 ms.
...Using existing profile dir: u'/home/ian/.config/ipython/profile_default'
...Completed registration with id 4
```

让我们测试配置。在例 10-8 中我们会使用新画像启动一个本地 IPython shell。我们会获取 5 个客户端列表（4 个本地，1 个远程），接着我们会请求 Python 的版本信息——我们能看到在远程机器上，我们正使用着 Anaconda 发布版。我们只得到了一个额外的引擎，因为在这个案例中，远程机器是一个单核的 MacBook。

例 10-8　测试新机器是集群的一部分

```
$ ipython --profile=mycluster
Python 2.7.5+ (default, Sep 19 2013, 13:48:49)
Type "copyright", "credits" or "license" for more information.
IPython 1.1.0—An enhanced Interactive Python.
...
In [1]: from IPython.parallel import Client

In [2]: c = Client()

In [3]: c.ids
Out[3]: [0, 1, 2, 3, 4]

In [4]: dview=c[:]

In [5]: with dview.sync_imports():
   ...:     import sys

In [6]: dview.apply_sync(lambda:sys.version)
Out[6]:
['2.7.5+ (default, Sep 19 2013, 13:48:49) \n[GCC 4.8.1]',
 '2.7.5+ (default, Sep 19 2013, 13:48:49) \n[GCC 4.8.1]',
 '2.7.5+ (default, Sep 19 2013, 13:48:49) \n[GCC 4.8.1]',
 '2.7.5+ (default, Sep 19 2013, 13:48:49) \n[GCC 4.8.1]',
 '2.7.6 |Anaconda 1.9.2 (64-bit)| (default, Jan 17 2014, 10:13:17) \n
          [GCC 4.1.2 20080704 (Red Hat 4.1.2-54)]']
```

让我们把这一切放在一起。在例 10-9 中，我们将使用 5 个引擎来估算 pi，就如我们在 10.6.1 节中所做的那样。这次我们将使用`@require`装饰器来在引擎中导入 random 模块。我们使用一个直接的视图来把我们的工作发送到引擎上，这会阻塞在那里直到所有的结果返回过来。接着我们就像以前所做的那样来估算 pi。

例 10-9　使用我们的本地集群估算 pi

```
from IPython.parallel import Client, require
NBR_ESTIMATES = 1e8

@require('random')
def calculate_pi(nbr_estimates):
    ...
    return nbr_trials_in_unit_circle

if __name__ == "__main__":
    c = Client()
    nbr_engines = len(c.ids)
    print "We're using {} engines".format(nbr_engines)
    dview = c[:]
    nbr_in_unit_circles = dview.apply_sync(calculate_pi, NBR_ESTIMATES)

    print "Estimates made:", nbr_in_unit_circles

    # work using the engines only
    nbr_jobs = len(nbr_in_unit_circles)
    print sum(nbr_in_unit_circles) * 4 / NBR_ESTIMATES / nbr_jobs
```

IPython Parallel 提供了比在这儿所展示的要多得多的功能。异步任务和在更大的输入区间上的映射当然是可能的。它也有一个`CompositeError`类，这是一个更高层次的异常，用来封装发生于多个引擎上的相同的异常（如果你部署了糟糕的代码，你不会接收到多个完全相同的异常！）。当你处理多个引擎时，这就是一个便利。

IPython Parallel 的一个尤其强大的特性就是允许你使用更大的集群环境，包括超级计算机和类似于亚马逊 EC2 的云服务。为了进一步方便这种类型的集群开发，Anaconda 发布版包含了对 StarCluster 的支持。Olivier Grisel 在 PyCon 2013 上给出了一个使用`scikit-learn`来做高级机器学习的一本优秀的教材。在两个小时内，他演示了使用 StarCluster 在亚马逊的 EC2 临时实例上通过 IPython Parallel 来做机器学习。

10.7 为鲁棒生产集群的 NSQ

在生产环境中，你需要远比我们所谈到的其他解决方案更健壮的解决方案。这是因为在你的集群每天运营期间，节点可能变得不可用，代码可能会崩溃掉，网络可能会断线，或者在其他数以千计会发生的问题中，其中的一个就可能发生了。问题就在于以前所有的系统有一台计算机来发布命令，还有有限和静态数量的计算机来读取并执行命令。我们宁愿用一个使用消息总线的多角色（actor）的系统来取而代之——这将允许我们具有数量任意和经常变化的消息创建者和消息消费者。

针对这些问题的一个简单解决方案就是 NSQ，一个高性能的分布式消息平台。尽管它是用 GO 编写的，但它是完全与数据格式和语言无关的。结果就是有很多语言写成的库，访问 NSQ 的基本接口是只需要能够创建 HTTP 调用的 REST API。而且，我们能够用想要的任何格式来传送消息：JSON、Pickel、`msgpack` 等。无论如何，最重要的是，它提供了关于消息递送的基本保证，并且它使用了两个简单的设计模式来做好了一切：队列和发布者/订阅者模式。

10.7.1 队列

队列是一种消息的缓存类型。无论何时，当你想把消息传送给处理管道的另一端时，你把它发送到队列，它会在队列里等待直到有可用的工作者来读取它。当生产和消费之间存在不平衡时，队列在分布式处理中是最有用的。如果发生了不平衡，我们仅仅通过添加更多的数据消费者即可水平扩展，直到消息生产的速率等于消费的速率。另外，如果负责消费消息的计算机下线了，消息不会丢失，只是在队列中排队，直到出现可用的消费者，这样就给了我们消息递送的保证。

例如，假设我们想要在用户每次给我们站点的商品评分的时候，给用户处理新的推荐。如果我们没有队列，那么“评分”的行为会直接调用“重新计算推荐”的行为，而不管服务器正拼命忙于处理推荐。如果突然间数以千计的用户决定给某件商品评分，我们的推荐服务器就可能会疲于应付这些请求，它们就可能会开始超时，丢弃消息，通常变得失去响应！

另一方面，当任务准备好时，推荐服务器使用队列来请求更多的任务。一个新的“评分”行为会把一个新任务放入队列，当推荐服务器准备做更多工作时，它会从队列中抓取任务来处理。在这个设定中，如果比平常更多的用户开始给商品评分，我们的队列将会塞满，对于推荐服务器来说它的行为就像是一个缓存——它们的工作负载将不受影响，它们还会处理消息，直到队列变空。

随之而来的一个潜在的问题就是如果队列完全被任务搞得不堪重负，它将会存储相当多的消息。NSQ 通过多个存储后端来解决这个问题——当没有许多消息时，它们保存在内存中；当更多的消息开始进来时，把消息放置进磁盘中。

备忘

一般来说，当使用队列系统来工作时，设法让下流的系统（例如，前面例子中的推荐系统）处于正常工作负载 60%的容量是一个好主意。在给问题分配足够多的资源与当工作量增加到超出正常水平时给你的服务器充足的额外能力之间进行权衡，这是一个良好的妥协。

10.7.2 发布者/订阅者

另一方面，pub/sub（发布者/订阅者的简称）描述了谁来得到哪些消息。数据发布者能够推送关于特定主题的数据，而数据订阅者注册不同的数据源。无论发布者何时发放信息，它都发送给所有的订阅者——它们各自得到原始信息的一份完全相同的拷贝。你可以把它想象成报纸：许多人能够订阅特定的报纸，无论新版本的报纸何时出来，每一个订阅者都得到一份完全相同的拷贝。另外，报纸的生产者完全不需要知道报纸要发送给的人群。结果就是，发布者和订阅者相互解耦了，当我们的网络发生变化，还处于生产环境中时，让我们的系统变得更健壮。

除此之外，NSQ 增加了数据消费者的概念，那就是，多个进程能够连接到相同的数据发布。无论新的数据何时出来，订阅者都得到一份数据拷贝。无论怎样，每个订阅只有一个消费者看到了数据。在与报纸的类比中，你可以把它想象成让多名阅读报纸的人处于相同的家庭中。发布者将把一份报纸递送到家中，既然家庭只订阅了一次，在家中谁先拿到报纸谁就可以阅读数据。当每一个发布者的消费者看到消息时，对消息做相同的处理。无论如何，它们可以悄悄地在多台计算机上，这样就更增强了整个计算池的处理能力。

我们在图 10-1 中可以看到对发布者/处理者模式的描述。如果一条关于“点击”主题的新消息发布出来了，所有的订阅者（或者，用 NSQ 的术语来说，就是通道——例如，“指标”、“作弊分析”，以及“打包”）将得到一份拷贝。每个订阅者由一个或多个消费者所组成，代表对响应消息的实际处理。在“指标”订阅者的情况下，只有一个消费者会看到新消息。下一条消息将到另一个消费者那去，依次类推。

在潜在的大规模的消费者池中传播消息的好处就是实质上做自动的负载均衡。如果一条消息要花费很长的时间来处理，消费者直到处理完成后，才会发信号给 NSQ 表示自己已准备好接受更多的消息，这样其他消费者将获得以后的大部

分消息（直到原来的消费者做好了再次处理的准备）。另外，它允许已经存在的消费者断开连接（无论是自主选择还是因为失效），还允许新的消费者连接到集群，然而又保持了在特定订阅组中的处理能力。例如，如果我们发现“指标”要花费相当长的时间来处理，而且常常跟不上需求，我们就能够仅仅为订阅组添加更多的进程给消费者池，以便给予我们更多的处理能力。另一方面，如果我们看到大多数进程处于空闲（例如，没有得到任何消息），我们能够轻易地从这个订阅组中移除消费者。

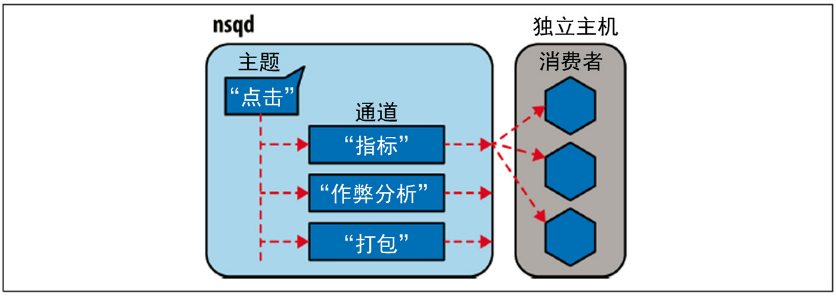


图 10-1　NSQ 的发布者/订阅者拓扑图

注意到无论是谁都能发布数据也很重要。消费者不仅仅一定是消费者——它可以从一个主题消费数据，接着发布另一个主题。事实上，当涉及分布式计算这种范式时，这条链是一个重要的工作流。消费者读取一个主题的数据，以某种方式转换数据，接着发布关于一个新主题的数据，而其他消费者能够进一步转换它。凭借这种方式，不同的主题代表不同的数据，订阅组代表对数据的不同转换，而消费者就是转换个体消息的实际工作者。

而且，在这个系统中存在极大的冗余。可以有许多 nsqd 进程让每个消费者连接上，可以有许多消费者连接到一个特定的订阅上。这样就没有单点失效问题，即使几台机器下线了，你的系统还是鲁棒的。我们可以看到在图 10-2 中，即使图表中的一台计算机下线了，系统还是能够投递和处理消息。另外，既然 NSQ 在关闭时把挂起的消息存储到了磁盘中，除非硬件失效是致命的灾难，否则你的数据还是非常有可能被完好无缺地投递。最后，如果消费者在响应一条特定的消息前关机了，NSQ 将会把消息重新发送给另外一个消费者。这意味着即使有多个消费者关机了，我们知道一个主题的所有消息将至少得到一次响应[①]。

① 当我们正使用 AWS 工作时，这将具有相当大的优势，我们能够让 nsqd 进程运行于一个保留实例上，而我们的消费者工作于临时实例的集群中。

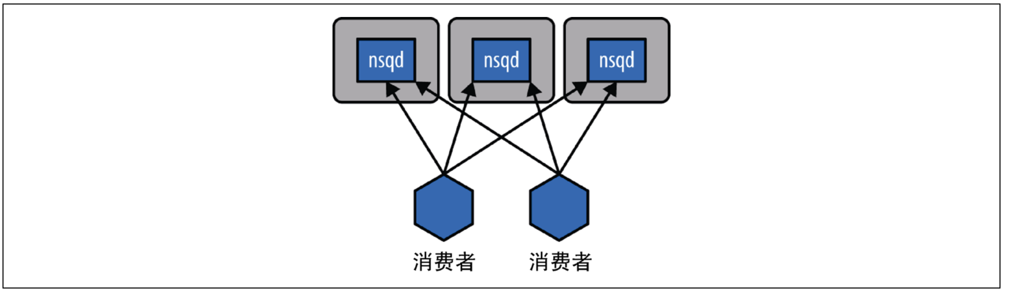


图 10-2　NSQ 的连接拓扑

10.7.3　分布式素数计算器

使用 NSQ 的代码一般是异步[①]的（请看第 8 章对它的完全解释），尽管它不一定非要如此。在下面的例子中，我们会创建一个工作者池，工作者读取一个称作 numbers 的主题，消息只是含有数字的 JSON 二进制对象。消费者将读取这个主题，查找数字是否为素数，接着根据数字是否为素数来写入另一个主题。这将给我们两个新主题，primes 和 non_primes，其他消费者就可以连接上来做更多的运算[②]。

就如我们之前所说的，像这样来做 CPU 密集型的工作有很多好处。首先，我们得到了完全鲁棒性的保证，这可能对这个项目有用，也可能无用。无论如何，最重要的是，我们得到了自动的负载均衡。这意味着如果一个消费者得到了一个花费特别长时间处理的数字，另一个消费者就会上手。

我们通过创建一个具有主题和所声明的订阅组（可以在例 10-10 的末尾看到）的 `nsq.Reader` 对象来创建一个消费者。我们也必须声明运行 `nsqd` 实例的位置（或者 `nsqlookupd` 实例，我们在本节中不会接触到）。另外，我们声明一个 `handler`，这只是一个函数，对来自主题的每一条消息，它都会被调用到。为创建一个生产者，我们创建了一个 `nsq.Writer` 对象，并声明了一个或更多的 `nsqd` 实例要写入的位置。这样就给了我们异步写入 `nsq` 的能力，只要声明主题名字和消息即可[③]。

例 10-10　使用 NSQ 的分布式素数计算

```
import nsq
from tornado import gen

from functools import partial
```

① 这种异步性来自于 NSQ 的协议以基于推送的方式来发送消息给消费者。这使得我们的代码能够在后台从 NSQ 的连接中异步读取，当发现消息时就唤醒。

② 这种类型的数据分析链被称作管道化，可以是一种有效的方法来高效地对相同的数据执行多种类型的分析。

③ 你也能手动使用一个 HTTP 调用轻易地发布一条消息。无论如何，这个 nsq.Writer 对象大大简化了错误处理。

```
import ujson as json

@gen.coroutine
def write_message(topic, data, writer):
    response = yield gen.Task(writer.pub, topic, data) # ❶
    if isinstance(response, nsq.Error):
        print "Error with Message: {}: {}".format(data, response)
        yield write_message(data, writer)
    else:
        print "Published Message: ", data

def calculate_prime(message, writer):
    message.enable_async() # ❷
    data = json.loads(message.body)

    prime = is_prime(data["number"])
    data["prime"] = prime
    if prime:
        topic = 'primes'
    else:
        topic = 'non_primes'

    output_message = json.dumps(data)
    write_message(topic, output_message, writer)
    message.finish() # ❸

if __name__ == "__main__":
    writer = nsq.Writer(['127.0.0.1:4150', ])
    handler = partial(calculate_prime, writer=writer)
    reader = nsq.Reader(
        message_handler = handler,
        nsqd_tcp_addresses = ['127.0.0.1:4150', ],
        topic = 'numbers',
        channel = 'worker_group_a',
    )
    nsq.run()
```

❶ 我们将异步地把结果写入一个新主题，如果因某种原因失败了就重新写。

❷ 通过在消息上使 async 生效，我们能够在处理消息时执行异步的操作。

❸ 使用 async-enabled 消息，我们在处理完消息时必须给 NSQ 发信号。

为了设置 NSQ 生态系统，我们将在本地机器上启动一个 nsqd 的实例：

```
$ nsqd
2014/05/10 16:48:42 nsqd v0.2.27 (built w/go1.2.1)
2014/05/10 16:48:42 worker id 382
2014/05/10 16:48:42 NSQ: persisting topic/channel metadata to nsqd.382.dat
```

```
2014/05/10 16:48:42 TCP: listening on [::]:4150
2014/05/10 16:48:42 HTTP: listening on [::]:4151
```

现在，我们可以启动我们所想要的数量的 Python 代码的实例（例 10-10）。事实上，我们可以让这些实例运行在其他计算机上，只要在 `nsq.Reader` 实例化中对 `nsqd_tcp_address` 的引用还是合法的。这些消费者将连接到 `nsqd`，并等待关于 numbers 主题的消息被发布出来。

数据发布到 numbers 主题上有很多种办法。既然了解到掌握一个系统的方式要经历一段长久的过程才理解怎样合适地处理它，我们将使用命令行工具来做。我们可以仅仅使用 HTTP 接口来给主题发布消息：

```
$ for i in `seq 10000`
> do
>  echo {\"number\": $i} | curl -d@- "http://127.0.0.1:4151/pub?topic=numbers"
> done
```

当这个命令开始运行时，我们以不同的数字在其中发布消息给 numbers 主题。同时，所有的生产者将开始输出状态消息，表明它们已经看见并处理了消息。另外，这些数字或发布给 primes 主题，或发布给 non_primes 主题。这允许我们有其他的数据消费者连接到这些主题中的任意一个来得到一个过滤我们原始数据的子集。例如，一个只需要素数的应用可以只连接到 primes 主题而总是有新的素数来为它自己的运算。我们可以通过使用 nsqd 的 `stats` HTTP 端点来观看我们运算的状态：

```
$ curl "http://127.0.0.1:4151/stats"
nsqd v0.2.27 (built w/go1.2.1)

  [numbers        ] depth: 0     be-depth: 0     msgs: 3060     e2e%:
    [worker_group_a ] depth: 1785  be-depth: 0      inflt: 1    def: 0
            re-q: 0     timeout: 0              msgs: 3060 e2e%:
       [V2 muon:55915  ] state: 3 inflt: 1      rdy: 0   fin: 1469
            re-q: 0     msgs: 1469     connected: 24s

  [primes        ] depth:  195  be-depth: 0    msgs: 1274       e2e%:

  [non_primes    ] depth: 1274  be-depth: 0    msgs: 1274       e2e%:
```

我们在这儿能看到 numbers 主题有一个订阅组 work_group_a 和一个消费者。另外，订阅组有一个长达 1785 条消息的大纵深，这意味着我们把消息放入 NSQ 的速度超过我们能够处理的速度。这个迹象表明要增加更多的消费者，这样我们就有更多的处理能力来应对更多的消息。而且，我们可以看到这个特定的消费者已经连接了 24 秒，已经处理了 1469 条消息，并且当前还有 1 条消息正在处理中。这个状态端点给出了大量信息来调试你的 NSQ 设置！最后，我们来看看 primes 和 non_primes 主题，它们没有订阅者或者消费者。这意味着消息将会存储起来直到有一个订阅者过来请求数据。

备忘

在生产系统中，你甚至能使用更加强大的工具 nsqadmin，它提供了一个 web 接口，具有很详细的关于所有主题/订阅者以及消费者的概况。另外，它允许你轻易地暂停以及删除订阅者和主题。

为了实际观看这些消息，我们将为 primes 主题（或者 non_primes）创建一个新消费者，只是把结果打包进一个文件或者数据库。或者，我们可以使用 nsq_tail 工具来看看数据包含了哪些内容：

```
$ nsq_tail --topic primes --nsqd-tcp-address=127.0.0.1:4150
2014/05/10 17:05:33 starting Handler go-routine
2014/05/10 17:05:33 [127.0.0.1:4150] connecting to nsqd
2014/05/10 17:05:33 [127.0.0.1:4150] IDENTIFY response:
                    {MaxRdyCount:2500 TLSv1:false Deflate:false Snappy:false}
{"prime":true,"number":5}
{"prime":true,"number":7}
{"prime":true,"number":11}
{"prime":true,"number":13}
{"prime":true,"number":17}
```

10.8 看一下其他的集群化工具

使用队列的任务处理系统自从计算机科学领域的开端以来就存在了，追溯到那个计算机很缓慢而且有许多任务需要被处理的时代。结果就是有许多队列库，其中很多能够在集群配置中使用。我们强烈推荐你挑选一个背后有积极社区的成熟库，支持你所需要的相同的特性集，并且没有太多的附加特性。

一个库具有越多的特性，则你会发现错误配置的情况也越多，从而在调试上浪费时间。当处理集群的解决方案时，简单化通常就是正确的目标。有一些使用更普遍的集群化解决方案：

- Celery（BSD 许可）是一个使用分布式消息架构的被广泛使用的异步任务队列，用 Python 所编写。它支持 Python、PyPy，以及 Jython。典型情况下它使用 RabbitMQ 作为消息代理，但是也支持 Redis、MongoDB 和其他的存储系统。它通常在 Web 开发项目中所使用。Andrew Godwin 在 12.6 节中讨论了 Celery。
- Gearman（BSD 许可）是一个多平台的任务处理系统。如果你正在使用不同的技术来集成处理任务，它是非常有用的。它具有对 Python、PHP、C++、Perl 以及其他许多语言的绑定。

- PyRes 是针对 Python 的基于 Redis 的轻量级的任务管理器。添加任务进 Redis 的队列中，设置消费者来处理它们，并且选择性地把结果在一个新的队列中传递回去。如果你的需求是轻量级的而且只用 Python，它是一个作为起点的非常简单的系统。

- 亚马逊的简单队列服务（SQS）是集成进亚马逊 Web 服务的一个任务处理系统。任务消费者和生产者能够存在于 AWS 内部或者外部，这样 SQS 启动简单，并且支持简单的迁移入云。对许多语言有库支持。

集群也能用于分布式的 `numpy` 处理，但是这是一个在 Python 世界中相对年轻的发展。通过 `distarrary` 和 `blaze` 包，Enthough 和 Continuum 都有解决方案。注意这些包企图为你处理同步化和数据局部性的复杂问题（没有一个适合所有情况的解决方案），所以要注意你可能会不得不思考你的数据布局和访取的方式。

10.9 小结

本书到此为止，我们已经看到了做剖析来理解你代码中运行慢速的部分，编译并使用 numpy 来让你的代码运行得更快，以及各种各样针对多进程和多主机的方法。在倒数第二章，我们将看到多种通过不同的数据结构和概率手段来使用更少 RAM 的方法。这些教程能够帮助你把所有数据存放于一台机器上，从而免去了运行集群的需求。

第 11 章

使用更少的 RAM

读完本章之后你将能够回答下列问题

- 为什么我应该使用更少的 RAM?
- 为什么 numpy 和 array 对存储大量数字而言更有利?
- 怎样把许多文本高效地存储进 RAM?
- 我该如何能仅仅使用一个字节来（近似地）计数到 1e77?
- 什么是布隆过滤？为什么我可能会需要它们?

我们很少会思考我们正在使用多少 RAM，一直到把它用完为止。如果你在扩展代码时用完了内存，它就会成为一个突如其来的阻碍者。把更多的东西纳入一台机器的 RAM 意味着更少的机器要管理，并且给你一条途径来为更大的项目规划容量。了解为什么 RAM 被吃光了而且考虑更有效的方式来使用这个稀缺资源将有助于你处理扩展性的问题。

另一种节约 RAM 的途径就是使用容器来利用你的数据特性进行压缩。在本章中，我们将看看 tries 树（有序的树数据结构）和 DAWG，后者能够把一个 1.1GB 的字符串集压缩到只有 254MB，而几乎不改变性能。第三种途径就是用空间来和准确性做交换。对于这种途径，我们将看看近似计数和近似集合成员，相比它们所对应的精确算法大大减少了 RAM 的使用。

对内存使用要考虑的一点就是“数据有质量”的观念。数据越多，移动起来就越慢。如果你能够吝啬于使用内存，你的数据将可能消耗得更快，因为它在总线上移动得更快，而且更多的数据将被纳入有限的缓存中。如果你需要把它存入离线

存储中（例如，一个硬盘驱动或者一个远程的数据集群），那么它会以慢得多的速度来传输进你的机器中。设法选择合适的数据结构，这样你的所有数据都能够纳入一台机器中。

对 Python 对象所使用的 RAM 量做统计棘手得令人吃惊。我们不必知道对象在幕后是如何被表示的，如果我们请求操作系统所使用的字节数，它将告诉我们分配给进程的总量。在这两种情况下，我们都不能精确地查看每个单独的 Python 对象占用的内存是怎样加入总量中的。

因为一些对象和库无法报告它们内部所分配的所有字节（或者它们包装了完全没有报告自己的内存分配的外部库），这应当是一种最佳猜测的情况。在本章中所探索的方法能够帮助我们决定最好的方式来表示我们的数据，从而整体上使用了更少的 RAM。

11.1　基础类型的对象开销高

使用存储着几百上千项的类似于 `list` 的容器来工作是普遍的。一旦你存储大数据，RAM 的使用就变成了一个问题。

一个具有 100000000 项的 list 大概要消耗 760MB，这是在假设所有条目都是相同对象的前提下。如果我们存储了 100000000 个不同项（例如，唯一的整数），那么我们得期望使用 GB 数量级的 RAM！每一个唯一的对象都有一个内存开销。

在例 11-1 中，我们在一个 `list` 中存储了许多 0 整数。如果你存储了 100000000 个任意对象的引用（无论对象的实例有多大），你还是期望要看见大约 760MB 的内存开销，因为 `list` 存储着对象的引用（不是对象的拷贝）。回头参考下 2.9 节来回忆怎样使用 `memory_profile`；在这里，我们使用 `%load_ext memory_profile` 来把它作为一个新的魔法函数载入进 IPython。

例 11-1　测量在一个 list 中的 100000000 个相同整数的内存使用

```
In [1]: %load_ext memory_profiler # load the %memit magic function
In [2]: %memit [0]*int(1e8)
peak memory: 790.64 MiB, increment: 762.91 MiB
```

对于下个例子，我们将从一个全新的 shell 开始。就如在例 11-2 中对 `memit` 的首次调用所揭示的那样，一个全新的 IPython shell 大约消耗了 20MB 的 RAM。接下来，我们可以创建一个具有 100000000 个唯一数字的临时 list。这总共大约消耗 3.1GB。

警告

在运行的进程中，内存能够被缓存起来，所以当使用 memit 来剖析时，先退出再重启 Python shell 总是更安全的方式。

在 `memit` 命令结束后，临时 list 被释放了。最终对 `memit` 的调用显示内存使用停留在大约 2.3GB。

问题

在读取答案之前，为什么 Python 进程还是可能要保持 2.3GB 的 RAM？在后台留下什么东西了吗，即使 list 已经到垃圾收集器中去了？

例 11-2　测量一个 list 中 100000000 个不同整数的内存使用

```
# we use a new IPython shell so we have a clean memory
In [1]: %load_ext memory_profiler
In [2]: %memit # show how much RAM this process is consuming right now
peak memory: 20.05 MiB, increment: 0.03 MiB
In [3]: %memit [n for n in xrange(int(1e8))]
peak memory: 3127.53 MiB, increment: 3106.96 MiB
In [4]: %memit
peak memory: 2364.81 MiB, increment: 0.00 MiB
```

100000000 个整数对象占据了 2.3GB 的绝大部分，即使它们不再被使用了。Python 缓存了类似整数的基础对象为以后所用。在一个 RAM 有限的系统中，这会造成问题，所以你应该注意到这些基础类型可能会构建在缓存中。

在例 11-3 中一个后续的 `memit` 创建了另一个含有 100000000 项的 list，消耗了大约 760MB，在这个回调期间总体占用了大约达到 3.1GB 的内存分配。760MB 单单为容器所用，因为底层的 Python 整数对象已经存在——它们在缓存中，这样就可以被复用。

例 11-3　再次测量在一个 list 中的 100000000 个不同整数的内存使用

```
In [5]: %memit [n for n in xrange(int(1e8))]
peak memory: 3127.52 MiB, increment: 762.71 MiB
```

接下来我们将看到我们能够使用 `array` 模块来以更为廉价的方式存储 100000000 个整数。

Array 模块以廉价的方式存储了许多基础对象

`Array` 模块高效地存储了类似于整数、浮点数和字符的基础类型，但没有复数或者类。它创建了一个连续的 RAM 块来保存底层数据。

在例 11-4 中，我们把 100000000 个整数（每个 8 字节）分配到一个连续的内存块。总体上，进程大约消耗了 760MB。这种方式和之前唯一整数列表的方式之间的差别是 `2300MB - 760MB == 1.5GB`。这是一个对 RAM 的巨大节约。

例 11-4　构建一个使用 760MB 的 RAM 的具有 100000000 个整数的数组

```
In [1]: %load_ext memory_profiler
In [2]: import array
In [3]: %memit array.array('l', xrange(int(1e8)))
peak memory: 781.03 MiB, increment: 760.98 MiB
In [4]: arr = array.array('l')
In [5]: arr.itemsize
Out[5]: 8
```

注意在 `array` 中的唯一数字不是 Python 对象，它们在 `array` 中是字节。如果我们要解引用它们中任何一个，那么一个新的 Python int 对象将会被构建。如果你想要在它们之上来做计算，不会发生整体上的节省，但是如果你想要把数组传递给一个外部进程或者只使用一些数据，你应该看到相比使用一个整数的 `list` 来说，大大节约了 RAM。

备忘

如果你正使用 Cython 在一个大数字数组或大数字矩阵上工作，并且你不想要对 `numpy` 的外部依赖，提醒你可以把你的数据存储在一个 `array` 中，并把它传进 Cython 来做处理，这样没有额外的内存开销。

`array` 模块使用一个有限的具有各种不同精度的 datatype 集（请看例 11-5）来工作。选择你需要的最小精度，这样你就会仅仅按需分配 RAM，而不是分配超出需求更多的 RAM。要注意字节的大小是平台相关的——这里的大小参考 32 位的平台（它声明了最小尺寸），而我们却是在一台 64 位的笔记本电脑上运行例子的。

例 11-5　由 array 模块所提供的基本类型

```
In [5]: array? # IPython magic, similar to help(array)
Type:       module
String Form:<module 'array' (built-in)>
Docstring:
This module defines an object type which can efficiently represent
an array of basic values: characters, integers, floating point
numbers. Arrays are sequence types and behave very much like lists,
except that the type of objects stored in them is constrained. The
type is specified at object creation time by using a type code, which
is a single character. The following type codes are defined:

    Type code   C Type             Minimum size in bytes
```

```
    'c'         character          1
    'b'         signed integer     1
    'B'         unsigned integer   1
    'u'         Unicode character  2
    'h'         signed integer     2
    'H'         unsigned integer   2
    'i'         signed integer     2
    'I'         unsigned integer   2
    'l'         signed integer     4
    'L'         unsigned integer   4
    'f'         floating point     4
    'd'         floating point     8

The constructor is:

array(typecode [, initializer]) -- create a new array
```

numpy 具有能够持有更广泛的 datatypes 的数组——你对每一项的字节的数量有更多的控制，并且你还可以使用复数和 datetime 对象。一个 complex128 对象采用每项 16 个字节：每项是一个 8 字节的浮点数对。你不能在一个 Python 数组中存储复杂的对象，但是它们在 numpy 中是自由使用的。如果你是一个 numpy 的新手，请回去看看第 6 章。

在例 11-6 中，你能看见 numpy 数组的另一个特性——你可以查询项的数量、每个基础类型的大小以及底层 RAM 块的组合存储总量。注意这不包括 Python 对象的开销（一般情况下，相比你存储在数组中的数据而言，这是微不足道的）。

例 11-6　在 numpy 数组中存储更多的复杂类型

```
In [1]: %load_ext memory_profiler
In [2]: import numpy as np
In [3]: %memit arr=np.zeros(1e8, np.complex128)
peak memory: 1552.48 MiB, increment: 1525.75 MiB
In [4]: arr.size # same as len(arr)
Out[4]: 100000000
In [5]: arr.nbytes
Out[5]: 1600000000
In [6]: arr.nbytes/arr.size # bytes per item
Out[6]: 16
In [7]: arr.itemsize # another way of checking
Out[7]: 16
```

使用一个常规的 list 在 RAM 中来存储许多数字比使用一个 array 对象要低效得多。应当发生更多的内存分配，每一次都花费时间。在更大的对象上也发生了运算，对缓存更不友好，整体上使用了更多的 RAM，这样一来可用于其他程序的 RAM 就更少了。

无论如何，如果你在 Python 的 `array` 内容上做任何工作，基础类型可能被转换成临时对象，抵消了它们的收益。当和其他进程通信时把它们当成数据存储来使用是 `array` 的一个很棒的使用场景。

如果你正在做重量级的数字运算，那么 `numpy` 数组几乎肯定是一个更好的选择，因为你得到了更多的 datatype 选项和许多专业而快速的函数。如果你想要让你的项目有更少的依赖性，你可能选择避开 `numpy`，尽管 Cython 和 Pythran 用 `array` 和 `numpy` 数组同样都工作得好。Numba 只用 `numpy` 数组来工作。

Python 提供了一些其他工具来理解内存使用，就如我们将要在下一节中看到的那样。

11.2 理解集合中的 RAM 使用

你可能想知道你是否能够请求 Python 关于每个对象所使用的 RAM 大小。Python 的 `sys.getsizeof(obj)` 调用会告诉我们一些关于对象所使用的内存情况（绝大多数而不是全部对象提供了这个调用）。如果你以前没有见过，那么提醒一下它不会给你所期待的关于容器的答案！

让我们通过查看一些基础类型来开始。在 Python 中的 `int` 是一个可变尺寸的对象，它起始于一个常规的整数，如果你计数超过 `sys.maxint`（在 Ian 的 64 位笔记本电脑上，这个值是 9223372036854775807），它就会转变为一个长整数。

作为一个常规的整数，它占用 24 字节（对象有许多开销）；而作为长整数，它消耗 36 字节：

```
In [1]: sys.getsizeof(int())
Out[1]: 24
In [2]: sys.getsizeof(1)
Out[2]: 24
In [3]: n=sys.maxint+1
In [4]: sys.getsizeof(n)
Out[4]: 36
```

我们可以对 byte string 做同样的检查。一个空字符串消耗 37 字节，每一个多加的字符增加了 1 字节的开销：

```
In [21]: sys.getsizeof(b"")
Out[21]: 37
In [22]: sys.getsizeof(b"a")
Out[22]: 38
In [23]: sys.getsizeof(b"ab")
```

```
Out[23]: 39
In [26]: sys.getsizeof(b"cde")
Out[26]: 40
```

当我们使用一个列表时，我们看见了不同的表现。getsizeof 没有对列表的内容计数，仅仅是列表自身的开销。一个空列表消耗 72 字节，在一台 64 位笔记本电脑上，列表中的每一项占用了 8 个额外字节：

```
# goes up in 8-byte steps rather than the 24 we might expect!
In [36]: sys.getsizeof([])
Out[36]: 72
In [37]: sys.getsizeof([1])
Out[37]: 80
In [38]: sys.getsizeof([1,2])
Out[38]: 88
```

如果我们使用 byte string，这就更明显了——我们期望你看到比 getsizeof 所报告的要大得多的开销。

```
In [40]: sys.getsizeof([b""])
Out[40]: 80
In [41]: sys.getsizeof([b"abcdefghijklm"])
Out[41]: 80
In [42]: sys.getsizeof([b"a", b"b"])
Out[42]: 88
```

getsizeof 只报告了一部分开销，常常仅仅是父对象的开销。就如之前所提示的那样，它也不总是被实现了的，所以可以作有限的用途。

一个轻量级的更好的工具是 asizeof，它会遍历容器的层级结构并对它所发现的每个对象做出最好的猜测，给整体大小增加了尺寸。注意它的速度相当慢。

除了依赖于猜测和假设之外，它也不能计算幕后的内存分配（例如，一个包装了 C 库的模块可能没有报告在 C 库中所分配的字节数）。最好把它用来作为一个指导。我们倾向于使用 memit，因为它给了我们在问题机器上的准确的内存使用计数。

你如下所示使用 asizeof：

```
In [1]: %run asizeof.py
In [2]: asizeof([b"abcdefghijklm"])
Out[2]: 136
```

我们可以检查它对一个大列表所做的估算——这里我们将使用 100000000 个整数：

```
# this takes 30 seconds to run!
In [1]: asizeof([x for x in xrange(10000000)]) # 1e7 integers
Out[1]: 321528064
```

我们可以通过使用 memit 来查看进程如何增长来验证这个估算。在这种情况下，数字是非常接近的：

```
In [2]: %memit([x for x in xrange(10000000)])
peak memory: 330.64 MiB, increment: 310.62 MiB
```

asizeof 一般要比使用 memit 来得更慢，但是当你分析小对象时，asizeof 是可以用上的。memit 可能对真实世界的应用来说更加有用，因为进程的实际内存使用是测量出来的，而不是推导出来的。

11.3 字节和 Unicode 的对比

转换到 Python 3.3+的一个令人信服的理由就是它对 Unicode 对象的存储明显要比在 Python 2.7 中少。如果你主要处理许多字符串，并且它们吃掉了很多 RAM，一定要考虑转移到 Python 3.3 中去。这样你就绝对免费地得到了对 RAM 的节省。

在例 11-7 中，我们可以看到 100000000 个字符的序列被构建成字符集合（这与在 Python 2.7 中的常规的 str 相同）以及被构建成 Unicode 的对象。Unicode 变体占用了多达 4 倍的 RAM，每一个 Unicode 字符耗费了相同的更高的代价，而不管表示底层数据所需的字节的数量是多少。

例 11-7 Unicode 对象在 Python 2.7 中是代价高昂的

```
In [1]: %load_ext memory_profiler
In [2]: %memit b"a" * int(1e8)
peak memory: 100.98 MiB, increment: 80.97 MiB
In [3]: %memit u"a" * int(1e8)
peak memory: 380.98 MiB, increment: 360.92 MiB
```

Unicode 对象的 UTF-8 编码为每个 ASCII 字符使用一个字节，而为更少见到的字符使用了更多的字节。Python 2.7 为每一个 Unicode 字符使用了相同数量的字节数，而不管字符的出现率。如果你对 Unicode 编码和 Unicode 对象的对比没有把握，那么请去看看 Net Batchelder 的“实践 Unicode，或者是，我该如何结束痛苦？”

从 Python 3.3 开始，多亏了 PEP 393，我们具有了灵活的 Unicode 表示。它通过观察字符串中的字符范围，并且尽可能使用更少的字节数来表示更低阶的字符的方式来工作。

在例 11-8 中，你可以见到字节的开销和 ASCII 字符的 Unicode 版本的开销是一样的，并且使用了非 ASCII 的字符（sigma）仅仅翻倍地使用了内存——这还是比 Python 2.7 中的境遇要更好。

例 11-8　Unicode 对象在 Python 3.3+中要远远更低廉

```
Python 3.3.2+ (default, Oct 9 2013, 14:50:09)
IPython 1.2.0 -- An enhanced Interactive Python.
...
In [1]: %load_ext memory_profiler
In [2]: %memit b"a" * int(1e8)
peak memory: 91.77 MiB, increment: 71.41 MiB
In [3]: %memit u"a" * int(1e8)
peak memory: 91.54 MiB, increment: 70.98 MiB
In [4]: %memit u"Σ" * int(1e8)
peak memory: 174.72 MiB, increment: 153.76 MiB
```

在 Python 3.3 默认 Unicode 对象的前提下，如果你在许多字符串数据上工作，你几乎肯定会受益于这个升级。缺少低廉的字符串存储对一些人来说是在早期 Python 3.1 期间遇到的一个障碍，但是现在随着 PEP 393，这完全不是一个问题。

11.4　高效地在 RAM 中存储许多文本

使用文本的一个普遍的问题就是它占用了许多 RAM——但是如果我们想要测试一下我们是否在之前见到过字符串或对它们的频率进行计数，那么让它们在 RAM 中是方便的，而不是让它们从磁盘中来回换页。以自然的方式存储字符串是代价昂贵的，但是 trie 树和有向无环的单词图（DAWGs）能够被用来压缩表示它们，然而又允许进行快速的操作。

这些更高级的算法能够让你显著节约 RAM 的使用量，这意味着你可能不需要扩展到更多的服务器上，对于生产系统有巨大的节省。在本节中，我们将看看使用 trie 树来压缩一个字符串集，从耗费 1.1GB 下降到 254MB，而仅让性能发生微小的变化。

对于这个例子，我们将从维基百科的部分转储上构建而来。这个集合包含了 8545076 个唯一的符号，来自于英语维基百科的一部分，在磁盘上占用了 111707546（111MB）之多。

符号从它们原来的文章上以空格符来分割。它们是具有可变长度的，并且包含了 Uncode 字符和数字。它们看起来就像：

```
faddishness
'melanesians'
Kharálampos
PizzaInACup™
url="http://en.wikipedia.org/wiki?curid=363886"
VIIIa),
Superbagnères.
```

我们将使用这个文本例子来测试我们如何能快速地构建持有每个唯一单词实例的数据结构，然后我们将看看如何能快速地查询已知的单词（我们将使用不常见的“Zwiebel”，来自于画家 Alfred Zwiebel）。这让我们发问，“我们以前曾讲过 Zwiebel 吗？”符号寻找是一个普遍的问题，要能够快速地做出来是重要的。



备忘

当你在你自己的问题上尝试这些容器时，要注意你可能会看到不同的表现。每种容器以不同的方式构建了自己内部的结构，传递不同类型的符号可能会影响结构的构建时间，而且不同长度的符号会影响查询时间。总是要以系统的方法来做测试。

在 800 万个符号上尝试这些方法

图 11-1 显示了使用一些容器存储了 800 万个符号的文本文件（111MB 原始数据），这些容器就是我们在本节中将要讨论的。*X* 轴显示了每个容器的 RAM 使用情况，*Y* 轴跟踪了查询时间，每个点的大小与构建结构所花的时间有关（更大的点意味着花费更久）。

就如我们在这张图表中所能见到的那样，`set` 和 `DAWG` 的例子使用了许多 RAM。List 的例子内存开销大而且又慢。Marisa trie 树和 HAT trie 树的例子对于这个数据集是最有效的，它们使用了其他方法四分之一的 RAM，而几乎没有改变查找速度。

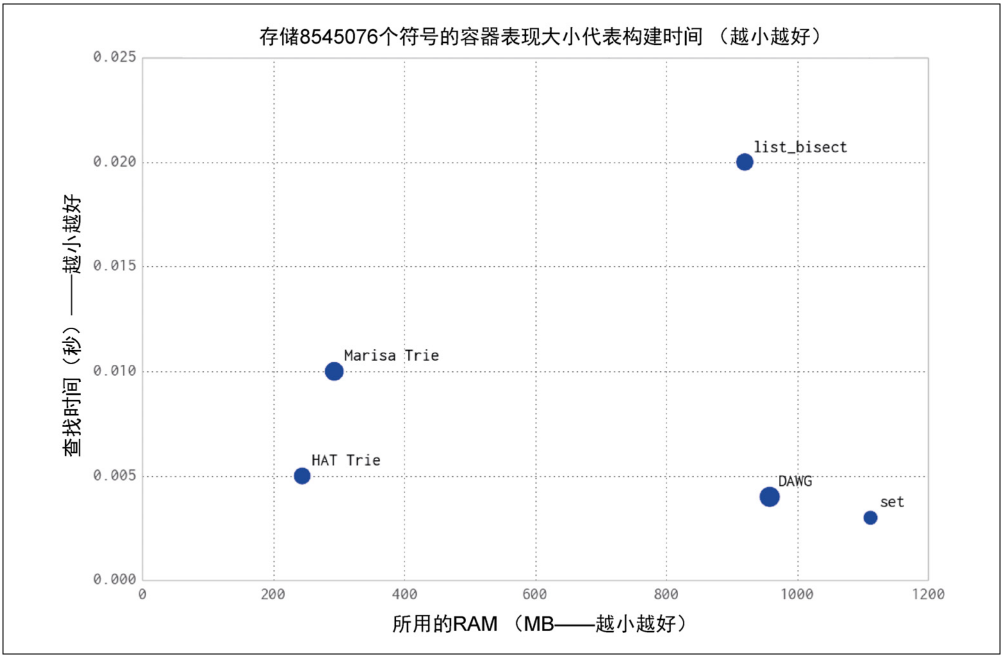


图 11-1　DAWG 和 tries 树与内置容器的对比

图表没有显示对没有排序方法的简单 list 的查找时间，我们将马上介绍，因为它花费得太久了。datrie 的例子没有在绘图中包含，因为它会产生一个段错误（我们已经在过去的另一项任务中有这个问题）。当它工作时，它是快速而紧凑的，但是它能够展现出无法控制的难以做出解释的构建时间。因为它能比其他方法更快，所以值得把它包括进来，但是显而易见，你将要在你的数据上彻底测试它。

请注意你必须要使用多种不同的容器来测试你的问题——每一个容器有不同的权衡之处，例如构建时间和 API 的灵活性。

接下来，我们将构建一个进程来测试每个容器的行为。

1．list

让我们从最简单的方法开始。我们会把我们的符号加载进一个 list 中，接着使用一个 O(n) 的线性搜索来查询它。我们不能在已经提到过的大型例子中这样做——搜索花费了太久的时间——所以我们将用一个小得多的例子（499048 个符号）来演示技巧。

在接下来的每一个例子中，我们都使用一个产生器 text_example.readers，用来同时从输入文件中抽取出 Unicode 符号。这意味着读取进程只使用了很少量的 RAM：

```
t1 = time.time()
words = [w for w in text_example.readers]
print "Loading {} words".format(len(words))
t2 = time.time()
print "RAM after creating list {:0.1f}MiB, took {:0.1f}s".
    format(memory_profiler.memory_usage()[0], t2 - t1)
```

我们对能够以多快的速度查询到这个列表感兴趣。理想情况下，我们想要找到一个容器来存储我们的文本并且允许我们没有任何代价地来查询和修改它。为了查询它，我们使用 timeit 来对已知的单词进行数次查找：

```
assert u'Zwiebel' in words
time_cost = sum(timeit.repeat(stmt="u'Zwiebel' in words",
                              setup="from __main__ import words",
                              number=1,
                              repeat=10000))
print "Summed time to lookup word {:0.4f}s".format(time_cost)
```

我们的测试脚本报告大约 59MB，被用来把原来 5MB 的文件作为一个列表来存储，查找时间是 86 秒：

```
RAM at start 10.3MiB
Loading 499048 words
RAM after creating list 59.4MiB, took 1.7s
Summed time to lookup word 86.1757s
```

把文本存储进一个没有排序的 list 明显是一个糟糕的主意。O(n)的查找时间是代价高昂的，内存使用也同样如此。这是所有情况中最糟糕的！

我们能够通过对list做排序以及通过bisect模块使用二分查找法来改进查找时间。对于将来的查询来说，这给了我们一个合理的下限。在例 11-9 中我们对排序列表所花费的时间进行计时。在这里我们切换到更大的有 854506 个符号的集合。

例 11-9 对为了使用 bisect 而做准备的排序操作进行计时

```
    t1 = time.time()
    words = [w for w in text_example.readers]
    print "Loading {} words".format(len(words))
    t2 = time.time()
    print "RAM after creating list {:0.1f}MiB, took {:0.1f}s".
    format(memory_profiler.memory_usage()[0], t2 - t1)
    print "The list contains {} words".format(len(words))
    words.sort()
    t3 = time.time()
    print "Sorting list took {:0.1f}s".format(t3 - t2)
```

接下来我们像以前一样做相同的查找，但是使用额外的 index 方法，它使用了bisect:

```
import bisect
...
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect.bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError
...
    time_cost = sum(timeit.repeat(stmt="index(words, u'Zwiebel')",
                                  setup="from __main__ import words, index",
                                  number=1,
                                  repeat=10000))
```

在例 11-10 中，我们看到 RAM 的使用比以前要大得多，因为我们明显装载进了更多的数据。排序进一步用去了 16 秒，而累计的查找时间却是 0.02 秒。

例 11-10 对在一个排序列表上使用 bisect 进行计时

```
$ python text_example_list_bisect.py
RAM at start 10.3MiB
Loading 8545076 words
```

```
RAM after creating list 932.1MiB, took 31.0s
The list contains 8545076 words
Sorting list took 16.9s
Summed time to lookup word 0.0201s
```

现在我们有了一个对字符串查找计时的合理的基线：RAM 使用必须要好于 932MB，并且总共查找时间应该比 0.02 秒更少。

2．set

使用内置的 set 看起来或许是处理我们任务的最明显的方法了。在例 11-11 中，set 在一个散列结构中存储了每一个字符串（如果你需要清醒一下，请看第 4 章）。检查成员是快速的，但是每一个字符串必须分开存储，在 RAM 上代价昂贵。

例 11-11　使用一个 set 来存储数据

```
words_set = set(text_example.readers)
```

就如我们在例 11-12 中所见的那样，set 比 list 使用了更多的 RAM；无论如何，它给了我们一个非常快速的查询时间，而不需要一个额外的 index 函数或者一个中间的排序操作。

例 11-12　运行 set 的例子

```
$ python text_example_set.py
RAM at start 10.3MiB
RAM after creating set 1122.9MiB, took 31.6s
The set contains 8545076 words
Summed time to lookup word 0.0033s
```

如果 RAM 不昂贵，那么这可能是最合理的首选方法。

然而，我们现在已经丧失了原来数据的顺序。如果顺序对你是重要的，提示你可以把字符串作为键存储在字典里，每个值就是和原来的读取顺序相关的索引。这样，你就能向字典查询键是否存在，并请求它的索引。

3．更有效的树结构

让我们介绍一组更高效地使用 RAM 来表示字符串的算法。

来自维基共享资源的图 11-2 展示了 4 个单词在 trie 树和 DAWG[①] 中的不同表示："tap"、"taps"、"top" 和 "tops"。DAFSA 是 DAWG 的另一个名称。使用一个 list 或者 set，这些单词中的每一个都作为一个独立的字符串存储。DAWG 和 trie 树共享了字符串的公共部分，所以用到的 RAM 更少。

① 这个例子取自关于确定性无环有限状态机（DAFSA）的维基文章。DAFSA 是 DAWG 的另一个名称。附随的图片来自于维基共享资源。

DAWG 和 trie 树的主要区别是 trie 树只共享公共前缀，然而 DAWG 共享公共前缀和后缀。在有很多公共单词前缀和后缀的语言中（就像英语），这样能减少很多重复。

精确的内存表现取决于你的数据结构。典型说来，一个 DAWG 不能分配一个值给键，这归因于从字符串的开始到结束之间的多条路径，但是展示在这里的版本能够接受一个值映射。Trie 树也能接收一个值映射。有些结构不得不在开始扫描时构建，其他的则能在任何时候更新。

这些结构的一个巨大优势就是它们提供了一个公共前缀搜索，那就是你可以请求与你提供的前缀相同的所有单词。使用我们的 4 个单词列表，当搜索“ta”时，结果将是“tap”和“taps”。而且，既然这些结果是通过图结构所发现的，获取到它们是很快速的。例如，如果你正工作于 DNA 方面，使用 trie 树编辑数百万的短单词是减少 RAM 使用的一种有效方式。

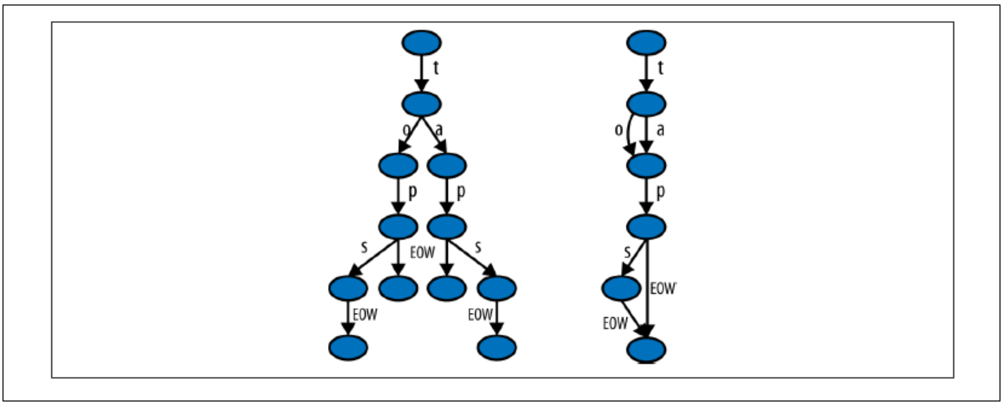


图 11-2　Trie 树和 DAWG 结构（图片由 Chkno 提供【CC BY-SA 3.0】）

在下面的小节中，我们凑近看一看 DAWGs、trie 树和它们的用途。

4．有向无环单词图（DAWG）

有向无环图（MIT 授权）企图高效地表示共享公共前缀和后缀的字符串。

在例 11-13 中，你会看到对 DAWG 的很简单的设置。对于这个实现，DAWG 在创建后不能被修改，它读取了一个迭代器来立即创建自己。缺少创建之后的更新可能对你的使用场景来说是一个败笔。如果是这样，你可能需要研究用 trie 树来取代它。DAWG 的确支持丰富的查询，包括前缀查找。它也允许持久化，并且支持把整数索引作为值与字节及记录值一起存储起来。

例 11-13　使用 DAWGl 来存储数据

```
import dawg
...
    words_dawg = dawg.DAWG(text_example.readers)
```

就如你能够在例 11-14 中所看见的那样，对于相同组的字符串集，它比之前的 set 例子仅仅稍微少用了一些 RAM。更类似的输入文本将会产生更强大的压缩。

例 11-14　使用 DAWG 的例子

```
$ python text_example_dawg.py
RAM at start 10.3MiB
RAM after creating dawg 968.8MiB, took 63.1s
Summed time to lookup word 0.0049s
```

5．Marisa trie 树

Marisa trie 树（LGPL 和 BSD 双授权）是一个使用 Cython 绑定外部库的静态 trie 树。因为它是静态的，它在创建之后就不能改动。它像 DAWG 一样支持把整数索引作为值与字符值及记录值一起存储。

一个键能被用来查询一个值，反之亦然。所有共享了相同前缀的键能被高效地找到。tries 树的内容可以被持久化。例 11-15 图示了使用一个 Marisa trie 树来存储我们的样本数据。

例 11-15　使用 Marisa trie 树来存储数据

```
import marisa_trie
...
    words_trie = marisa_trie.Trie(text_example.readers)
```

在例 11-16 中，我们可以看到相比 DAWG 的例子，在 RAM 存储方面有显著的提高，但是整体搜索时间更慢一点。

例 11-16　运行 Marisa trie 树例子

```
$ python text_example_trie.py
RAM at start 11.0MiB
RAM after creating trie 304.7MiB, took 55.3s
The trie contains 8545076 words
Summed time to lookup word 0.0101s
```

6．Datrie 树

双数组 trie 树，或者 datrie（LGPL 授权），使用了预先构建的字母表来高效地存储键。这种 trie 树能够在创建之后修改，但是只使用相同的字母表。它也能够寻找与所提供的键共享前缀的所有键，并且支持持久化。

它与 HAT trie 树一起提供了最快的查找时间之一。

> **备忘**
>
> 当使用 datrie 树来做维基上的例子以及做以往的 DNA 表示的工作时，它有一个变态的构建时间。与其他用几秒完成构建的数据结构相比，它可能要花费几分钟或数小时来表示 DNA 字符串。

datre 树需要一个字母表来呈现给构造函数，并且只有键才允许使用这个字母表。在我们的维基的例子中，这意味着我们需要对原始数据做两遍扫描。你可以在例 11-17 中看到这种情况。第一遍扫描把一个字母表的字符集构建成一个 set，第二遍扫描构建 trie 树。这个更慢的构建过程允许更快的查找时间。

例 11-17　使用一个双数组 trie 树来存储数据

```
import datrie
...
    chars = set()
    for word in text_example.readers:
        chars.update(word)
    trie = datrie.BaseTrie(chars)
...
    # having consumed our generator in the first chars pass,
        we need to make a new one
    readers = text_example.read_words(text_example.SUMMARIZED_FILE) # new generator
    for word in readers:
        trie[word] = 0
```

好悲伤，在这个例子数据集上，datrie 树抛出了一个段错误，所以我们不能显示你的计时信息。我们选择把它包含进来，因为在其他的测试中，它要比下面的 HAT Trie 树倾向于更快一点（但是 RAM 使用要低效一点）。我们已经使用它成功地进行了 DNA 搜索，所以如果你有一个静态的问题并且它可以工作，你可以对它能很好地工作具有信心。如果你的问题具有可变化的输入，无论如何，这可能不是一个合适的选择。

7．HAT trie 树

HAT trie 树（MIT 授权）使用了缓存友好的表达方式从而在现代 CPU 上达成非常快速的查找。它能够在创建后被修改，但是另一方面只有非常有限的 API。

对于简单的使用场景，它具有很棒的性能，但是 API 的局限性（例如，缺少前缀查找）可能让它对你的应用来说用途更少。例 11-18 演示了在我们的例子数据集上使用 HAT trie 树。

例 11-18　使用 HAT trie 树来存储数据

```
import hat_trie
...
    words_trie = hat_trie.Trie()
    for word in text_example.readers:
        words_trie[word] = 0
```

就如你能在例 11-19 中所见的那样，HAT trie 树提供了我们的新数据结构中最快的查找时间，以及卓越的 RAM 使用。它在 API 上的局限意味着它的使用受到了限制，但是如果你只是需要在许多字符串中快速查找，那么这可能是你的解决方案。

例 11-19　运行 HAT trie 例子

```
$ python text_example_hattrie.py
RAM at start 9.7MiB
RAM after creating trie 254.2MiB, took 44.7s
The trie contains 8545076 words
Summed time to lookup word 0.0051s
```

8．在生产系统中使用 tries 树 （和 DAWGs）

trie 树和 DAWG 数据结构提供了良好的收益，但是你还是必须在你的问题上对它们做基准测试，而不是盲目地采用它们。如果你的字符串上有重叠的序列，那么有可能你会看到对 RAM 使用的改进。

Tries 树和 DAWGs 不是那么知名，但它们在生产系统上却提供了强大的收益。我们在 12.4 节中有一个令人印象深刻的成功故事。在 DapApps（坐落在 UK 的一个 Python 软件工作室）的 Jamie Matthews 也有一个关于在客户系统中使用 trie 树来为客户做出更高效和更廉价的部署的故事：

> 在 DabApps，我们经常设法处理复杂的技术架构问题，通过把复杂问题划分为小的、自包含的组件，一般使用 HTTP 在组件间进行网络通信来做到。这种方式（被称为“面向服务的”或者“微服务”架构）有各种益处，包括有可能在多个项目间复用或者共享单个组件的功能。
>
> 一个这样的任务就是做邮编的地理编码，这常常是我们面向客户的项目中的需求。这个任务把全部的 UK 邮编（例如：“BN11AG”）转变成一个经纬度的坐标对，从而使得应用可以执行诸如距离测量之类的地理空间计算。
>
> 在它的最底层，一个地理编码数据库是一个简单的字符串之间的映射，能够从概念上被表示成一个字典。字典的键是邮编，以一个规范的形式存储（“BN11AG”），值就是坐标的某种表示（我们使用了地理散列编码，但是

简洁明了地来说，想象一下由逗号分隔的一对，例如：“50.822921，-0.142871”)。

在 UK 大约有 170 万个邮编。就如上面所描述的那样，简单地把全部数据集装载进一个 Python 字典要使用好几百兆字节的内存。使用 Python 的简单序列化（pickle）格式把这个数据结构在磁盘上做持久化需要一个不可接受的大量的存储空间。我们知道我们可以做得更好。

我们实验了几种不同的内存和磁盘存储以及序列化格式，包括把数据存储在诸如 Redis 和 LevelDB 之类的外部数据库中，而且还对键/值对做压缩。最终我们偶然想起使用 trie 树的主意。Trie 树在表示内存中的大量字符串方面极其高效，而且可利用的开源库（我们选择了“marisa-trie 树”）让它们变得非常易于使用。

最终的应用程序，包括了一个小型的由 Flask 框架所构建的 web API 使用了仅仅 30MB 的内存来表示完整的 UK 邮编数据库，并且能够令人愉快地处理大量的邮编查找请求。代码是简单的，服务很轻量化，而且无痛部署和运行在诸如 Heroku 一样的免费主机平台上，而且不需要或者不依赖于外部数据库。我们的实现是开源的，在 *https://github.com/j4mie/postcodeserver/* 上有提供。

——Jamie Matthews

DabApps.com 技术总监（UK）

11.5 使用更少 RAM 的窍门

一般来说，如果你能够避免把它放进 RAM，就去做。你所加载的每样东西都耗费你的 RAM。你可能会加载你的部分数据，例如使用一个内存映射；或者，你可能会使用生成器来加载你所需的部分数据，为了局部计算，而不是把它一次性全部加载进来。

如果你正用数字型的数据来工作，那么你几乎肯定会想要转而使用 `numpy` 数组——该包提供了许多直接工作在底层基本类型对象上的快速算法。与使用数字的列表相比，RAM 上的节省是巨大的，而且时间上的节省也一样令人惊奇。

在本书中你已经注意到我们一般使用 `xrange`，而不是 `range`，只是因为（在 Python 2.x 中）`xrange` 是一个产生器，然而 `range` 却构建了一个完整的列表。

构建一个 100000000 个整数的列表仅仅用来遍历正确的次数是过分的——RAM 耗费巨大而且完全不必要。Python 3.x 把 `range` 变成了一个产生器，这样你不再需要做这种改变。

如果你正使用字符串工作，并且你用的是 Python 2.x，如果你想要节约 RAM，设法坚持使用 `str` 而不是 `unicode`。如果你贯穿整个程序需要许多 Uncode 对象，你可能通过简单地升级到 Python 3.3+，就能受到更好的服务。如果你正要在一个静态结构中存储大量的 Uncode 对象，那么你可能想要调查下我们刚才讨论过的 DAWG 和 trie 树结构。

如果你正用许多比特字串来工作，调查下 `numpy` 和 `bitarray` 包，它们都有把比特打包进字节的高效表示。你可能也会受益于查看 Redis，它提供了高效的比特模式存储。

PyPy 项目正在试验更高效的同质数据结构的表示，这样相同的基本类型（例如，整数）的长列表在 PyPy 中要比在 CPython 中的等价结构体可能耗费要少得多。Micro Python 项目会让任何工作于嵌入式系统的人产生兴趣。它是一个缩微内存印记的，试图兼容于 Python 3 的 Python 实现。

这（几乎！）不用我说，你要知道当你设法优化 RAM 使用时，你必须要做基准测试，并且在你改变算法前，有一个适当的单元测试集会产生一个优厚的报酬。

已经回顾了几种高效地压缩字符串和存储数字的方法后，我们现在将看看以精度换取存储空间。

11.6　概率数据结构

概率数据结构允许你以精度来换取大幅度的内存使用下降。除此之外，你能在它们之上所做的操作数量比 `set` 或者 trie 树要有限得多。例如，使用一个单独的耗费 2.56KB 的 HyperLogLog++结构，你能够计数唯一项的数量一直到大约 7900000000 项，而具有 1.625%的误差。

这意味着如果我们尝试计数唯一的汽车牌照数字，如果我们的 HyperLogLog++计数器得出有 654192028，我们就会置信实际的数目在 643561407 到 664822648 之间。而且，如果这个精度不够，你可以仅仅给结构增加更多的内存，它就会做得更好。给它 40.96KB 的资源就会让误差从 1.625%降低到 0.4%。无论如何，即使假设没有开销，把这数据存储到一个 `set` 中也会耗费 3.925GB！

另一方面，HyperLogLog++结构将只能够对一个集合的牌照进行计数，并合并另一个集合的牌照。如此这般，我们就能够让每一个州有一个这样的结构，查找在那些州中有多少独立的牌照，然后再把它们合并起来得到整个国家的计数。如果给我们一个牌照，我们无法以非常优秀的准确度来告诉你我们以前是否见过它，而且我们无法给你一个我们曾经见过的牌照样本。

当你已经花费了时间去理解问题并且需要投入生产来能够回答关于一个非常巨大的数据集上的一个很小的问题集时，概率数据结构是奇妙的。每种不同的数据结构有它能够以不同的精度来回答的不同问题，所以找到合适的数据结构只和理解你的需求相关。

几乎在所有情况下，概率数据结构通过找到对数据的另一种表示形式来工作，这种表示更紧凑并且包含与回答某个问题集相关的信息。可以把这看成一种有损压缩的类型，我们可能丢失了数据的某些方面，但是却保留了必要的成分。既然我们允许数据丢失掉与我们所关心的特定问题集不一定相关的部分，这种有损压缩能够比我们之前使用 trie 树所看到的无损压缩高效得多。正因为如此，你选择使用哪种概率数据结构是相当重要的——你想要挑选为你的使用场景保留了合适信息的那一个！

在我们深入探索之前，应该澄清这里的“误差率”由标准差来定义。这个术语源自于描述高斯分布，说明函数围绕一个中心值的发散程度。当标准差增长时，值的数量就更偏离中心点。概率数据结构的误差率因此成名就是因为围绕着它们的所有分析都是基于概率的。如此这般，当我们说 HyperLogLog++算法有一个 $err=\frac{1.04}{\sqrt{m}}$ 的误差时，我们意思是指有 66%的机会误差会小于 *err*，有 95%的机会小于 2**err*，有 99.7%的机会小于 3**err*。[①]

11.6.1 使用 1 字节的 Morris 计数器来做近似计数

我们将介绍使用其中最早期的一种概率计数器——Morris 计数器（以 NSA 和贝尔实验室的 Robert Morris 来命名）来做概率计数的主题。应用包括在 RAM 受限的环境中（例如，在一台嵌入式计算机上）对数百万个对象进行计数，理解大数据流，以及类似图像和语音识别之类的问题。

Morris 计数器跟踪一个指数并以 $2^{exponent}$ 来对计数状态建模（不是一个正确的计数）——它提供了一个数量级的估计。这个估计使用概率规则来更新。

① 这些数字来自于高斯分布的 66-95-99 规则。更多的信息能够在维基百科条目中找到。

我们开始把指数设为 0。如果我们请求计数器的值，我们会给出 pow(2,exponent) =1（敏锐的读者会注意到这偏差一个——我们的确说过这是近似计数器!)。如果我们让计数器自增，它将会产生一个随机数（使用均匀分布），并且它会测试是否 random(0,1) <= 1 / pow(2, exponent)，对（pow(2, 0) == 1）始终为真。计数器自增，并把指数设为 1。

我们让计数器第二次自增，它就会测试是否 random(0, 1) <= 1 / pow(2, 1)。这将会有 50%的机会为真。如果测试通过，那么指数就递增了。如果不通过，那么对于这次递增请求，指数不递增。

表格 11-1 显示了对于每一个首指数，递增发生的几率。

表 11-1　　Morris 计数器细节

exponent	pow(2, exponent)	P(increment)
0	1	1
1	2	0.5
2	4	0.25
3	8	0.125
4	16	0.0625
…	…	…
254	2.894802e+76	3.454467e-77

为指数使用一个单独的无符号字节，我们能够近似计数的最大值是 math.pow(2, 255) == 5e76。当总数增大时，相对实际数量的误差就会变大，但是节省下来的 RAM 多得惊人，因为我们只使用了 1 字节，否则我们只好用 32 位的无符号多字节。例 11-20 展示了 Morris 计数器的一个简单实现。

例 11-20　简单的 Morris 计数器实现

```
from random import random

class MorrisCounter(object):
    counter = 0
    def add(self, *args):
        if random() < 1.0 / (2 ** self.counter):
            self.counter += 1

    def __len__(self):
        return 2**self.counter
```

使用这个例中的实现，我们能够看到在例 11-20 中，第一个递增计数器的请求成功了，而第二个失败了。[①]

① 一个更加完全崭新的使用了一个字节数组来创建多计数器的实现在 *https://github.com/ianozsvald/morris_counter* 中可用。

例 11-21　Morris 计数器库的例子

```
In [2]: mc = MorrisCounter()
In [3]: print len(mc)
1.0
In [4]: mc.add() # P(1) of doing an add
In [5]: print len(mc)
2.0
In [6]: mc.add() # P(0.5) of doing an add
In [7]: print len(mc) # the add does not occur on this attempt
2.0
```

在图 11-3 中，粗黑线显示了一个通常的在每一次迭代中递增的整数。在一个 64 位的计算机上，这是一个 8 字节的整数。三个 1 字节的 Morris 计数器的演进以点线来显示；*Y* 轴显示它们的值，近似表示了每次迭代中的真实的数量。展示了三个计数器来给你一个关于它们的不同轨迹以及总体趋势的概念。三个计数器完全相互独立。

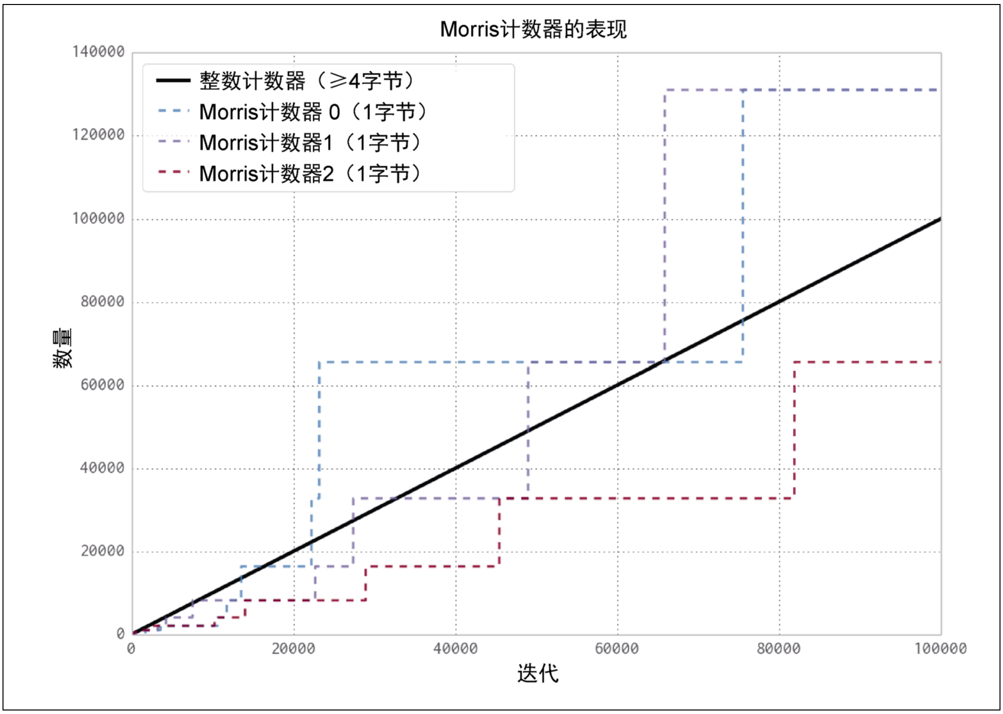


图 11-3　三个 1 字节的 Morris 计数器对比一个 8 字节的整数

这张图表给你一些使用 Morris 计数器时所要期望的误差概念。关于误差表现的进一步细节在线上有提供。

11.6.2 K 最小值

在 Morris 计数器中，我们丢失了所插入的项目的任何类型的信息。那就是说，计数器的内部状态是相同，而不管我们是`.add("micha")`还是`.add("ian")`。额外的信息是有用的，如果使用得当，能够帮助我们的计数器只对唯一项进行计数。这样做的话，调用`.add("micha")`几千次将只会增加计数器一次。

为了实现这种行为，我们将探索散列函数的属性（请看 4.1.4 节来得到对散列函数的更深入的讨论）。

我们想要利用的主要属性就是基于这样一个事实：散列函数获取输入并且均匀地分配它。例如，让我们假设有一个散列函数获取一个字符串并输出一个 0 到 1 之间的数字。那个函数均匀分布意味着当我们给它输入一个字符串时，我们以与得到一个 0.2 的值相等的几率来得到一个 0.5 的值，或者任何一个其他的值。这也意味着如果我们给它输入很多字符串值，我们将期望这些值被相对均匀排布。记住，这是一个概率化的论点：值不是总会被均匀排布，但是如果我们有许多字符串，并且尝试了这个实验很多次，它们会趋向于均匀地排布。

假设我们获取了 100 项并存储了那些值的散列（散列从 0 到 1 数数）。知道均匀排布意味着与其说"我们有 100 项"，不如说"我们每一项之间的间隔是 0.01"。这是 K 最小值算法[①]最终的出处——如果我们保存了我们所见到的 `k` 个最小的唯一散列值，我们就能近似估算散列值的总体空间，并且推导出这些项的全部数量。在图 11-4 中，当越来越多的项加进来时，我们能看见一个 K 最小值结构（也叫作一个 KMV）的状态。首先，既然我们没有很多散列值，我们保存的最大散列就是相当大的。当我们把越来越多项加进来时，我们所保存的最大的 `k` 个散列值就会变得越来越小。使用这个方法，我们能够得到 $O\left(\frac{2}{\pi(k-2)}\right)$ 的误差率。

`k` 越大，我们就越能说明我们所使用的散列化函数对于我们的特定输入不是完全均匀的，而且还是造成不幸散列值的原因。一个不幸散列值的例子就是对`['A', 'B', 'C']`做散列化，得到了值`[0.01, 0.02, 0.03]`。如果我们开始散列化越来越多的值，它们会聚合起来的可能性就越来越少。

而且，既然我们只保留最小的唯一散列值，数据结构只考虑唯一的输入。我们早就能看到这个，因为如果我们处在一个只存储了最小的三个散列并且当前`[0.1, 0.2,`

① Beyer,K.,Haas, P.J., Reinwald, B., Sismanis, Y.,和 Gemulla, R."在多集和操作下，有关对独立值估算的概要"。2007 年 ACM SIGMOD 数据管理国际会议进展——SIGMOD '07, (2007): 199 210. doi:10.1145/1247480.1247504。

0.3]是最小的散列的状态中，如果我们增加了任何散列值是 0.4 的项，我们的状态将保持不变。同样，如果我们添加了散列值是 0.3 的更多项进来，我们的状态也不会改变。这是一个叫作幂等性的属性，它意味着如果我们对这个结构多次用相同的输入做相同的操作，状态将保持不变。这与某些结构相反，例如，在一个 list 的尾部添加总是会改变它的值。幂等性的概念贯穿于本小节中的所有数据结构，但是除了 Morris 计数器之外。

例 11-22 展示了一个非常基本的 K 最小值实现。值得注意的是我们使用了一个 sortedset，它就像一个 set，但是只包含唯一项。这种唯一性免费地把幂等性给了我们的 KMinValues 结构。为了看到它，请跟随代码：当相同的项被加入了超过一次，数据属性不发生改变。

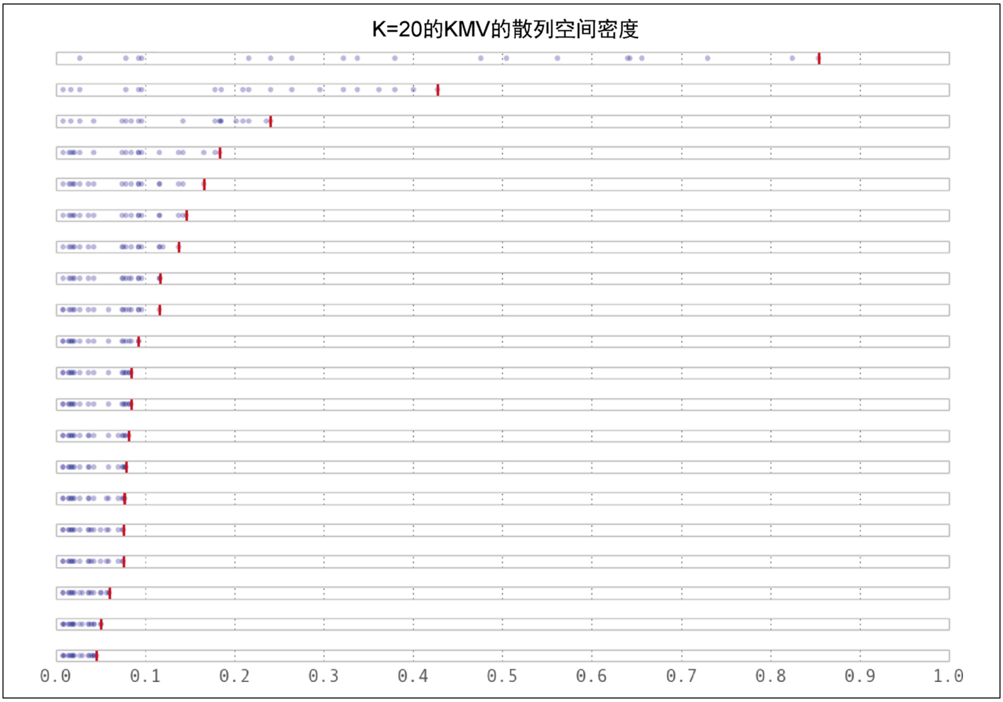


图 11-4　当更多的元素加进来时，值存储在一个 KMV 结构中

例 11-22　简单的 KMin 值实现

```
import mmh3
from blist import sortedset

class KMinValues(object):
    def __init__(self, num_hashes):
        self.num_hashes = num_hashes
```

```
        self.data = sortedset()

    def add(self, item):
        item_hash = mmh3.hash(item)
        self.data.add(item_hash)
        if len(self.data) > self.num_hashes:
            self.data.pop()

    def __len__(self):
        if len(self.data) <= 2:
        return 0
    return (self.num_hashes - 1) * (2**32-1) / float(self.data[-2] + 2**31 - 1)
```

使用在 Python 的包 countmemaybe（例 11-23）中的 KMinValues 的实现，我们可以开始看看这个数据结构的用途。这个实现与例 11-22 中的非常类似，但是它完全实现了其他的集合操作，例如并集与交集。也要注意“大小”和“势”被交替使用（单词“势”来自于集合理论，更多的在分析概率数据结构中被用到）。在这里，我们能看见即使让 k 使用一个小得合理的值，我们也能够存储 50000 项，而且能够以相对低的误差计算许多集合操作的势。

例 11-23 countmemaybe 的 KMinValue 实现

```
>>> from countmemaybe import KMinValues

>>> kmv1 = KMinValues(k=1024)

>>> kmv2 = KMinValues(k=1024)

>>> for i in xrange(0,50000): #❶
    kmv1.add(str(i))
   ...:

>>> for i in xrange(25000, 75000): #❷
    kmv2.add(str(i))
   ...:

>>> print len(kmv1)
50416

>>> print len(kmv2)
52439

>>> print kmv1.cardinality_intersection(kmv2)
25900.2862992

>>> print kmv1.cardinality_union(kmv2)
75346.2874158
```

❶ 我们在 kmv1 中放入了 50000 个元素。

❷ 在 kmv2 中也放入 50000 个元素，其中 25000 个与在 kmv1 中的相同。

备忘

使用这些类型的算法，散列函数的选择对于估算的质量具有巨大的影响。这些实现都使用了 mmh3，Python 实现的 mumurhash3 对于散列化字符串具有良好的属性。无论如何，可以使用不同的散列函数，如果它们对你特定的数据集更方便的话。

11.6.3 布隆过滤器

有时我们需要能够做其他类型的集合操作，为此，我们需要引入新的概率数据结构类型。布隆过滤[1]被创造出来用以回答我们是否在之前看到过一个条目的问题。

布隆过滤器通过多散列值的方式来工作，用于把一个值表示成多个整数。如果我们之后看到了相同的整数集，我们就能相当确信这就是相同的值。

为了以高效利用可用的资源的方式做到这一点，我们暗暗地把整数编码成一个列表的索引。这可能被看作一个初始化成 False 的布尔型的列表。如果我们被要求增加一个散列值为[10, 4, 7]的对象，那么我们设置列表的第十、第四和第七个索引为 True。将来，如果我们被问到以前是否看见到一个特定的条目，我们仅仅查找它的散列值并检查布尔型列表中对应的点是否设置成了 True。

这种方法不会给我们带来假阴性，而会给我们带来一个可控的假阳性率。这意味着如果布隆过滤器说我们以前没有看见过一个条目，那么我们 100%确认我们的确以前没有见过该条目。另一方面，如果布隆过滤器说我们以前曾看见过一个条目，那么有一定的概率我们其实并没有看见过，并且我们只是看见了一个错误的结果。这个错误的结果来自于我们会有散列冲撞的事实，有时两个对象的散列值会是相同的，即使它们不是同一个对象。无论如何，在实践中，布隆过滤器被设置成具有不到 0.5%的误差率，所以这种错误是可以被接受的。

备忘

我们可以仅凭两个相互独立的散列函数来模拟具有我们想要的数量的散列函数。这个方法被称作“双散列”。如果我们有一个给出两个相互独立的散列值的散列函数，我们可以这样做：

```
def multi_hash(key, num_hashes):
    hash1, hash2 = hashfunction(key)
    for i in xrange(num_hashes):
        yield (hash1 + i * hash2) % (2^32 - 1)
```

① Bloom,B.H. “以允许的误差使用 hash 编码做空间/时间的权衡。”ACM 通信。13:7（1970）：422-426 doi:10.1145/362686.362692

模数确保了作为结果的散列值是 32 比特的（我们可以通过 `2^64 - 1` 来为 64 比特的散列函数求摸）。

我们所需的布尔型列表的准确长度以及每一个条目的散列值的数量将基于我们所要求的容量和误差率而定。使用一些相当简单的统计参数[①]，我们看到理想的值就是：

$$num_bits = -capacity \cdot \frac{\log(error)}{\log(2)^2}$$

$$num_hashes = num_bits \cdot \frac{\log(2)}{capacity}$$

那就是说，如果我们希望以 0.05%的假阳性（那就是说，当我们说曾经看到过一个对象时，有 0.05%的几率其实我们并没有见过它）来存储 50000 个对象（不管对象自己有多大），那就需要 791015 比特的存储空间和 11 个散列函数。

为了进一步提高我们使用内存的效率，我们能够使用单个比特来表示布尔值（一个本地的布尔型实际上占用 4 比特位）。我们能够通过使用 `bitarray` 模块来轻易地做到。例 11-24 展示了一个简单的布隆过滤器的实现。

例 11-24　简单的布隆过滤器实现

```
import bitarray
import math
import mmh3

class BloomFilter(object):
    def __init__(self, capacity, error=0.005):
        """
        Initialize a Bloom filter with given capacity and false positive rate
        """
        self.capacity = capacity
        self.error = error
        self.num_bits = int(-capacity * math.log(error) / math.log(2)**2) + 1
        self.num_hashes = int(self.num_bits * math.log(2) / float(capacity)) + 1
        self.data = bitarray.bitarray(self.num_bits)

    def _indexes(self, key):
        h1, h2 = mmh3.hash64(key)
        for i in xrange(self.num_hashes):
            yield (h1 + i * h2) % self.num_bits
```

① 维基关于布隆过滤器的页面上有一个非常简单的对布隆过滤器属性的证明。

```
    def add(self, key):
        for index in self._indexes(key):
            self.data[index] = True

    def __contains__(self, key):
        return all(self.data[index] for index in self._indexes(key))

    def __len__(self):
        num_bits_on = self.data.count(True)
        return -1.0 * self.num_bits *
                    math.log(1.0 - num_bits_on / float(self.num_bits)) /
                        float(self.num_hashes)

    @staticmethod
    def union(bloom_a, bloom_b):
        assert bloom_a.capacity == bloom_b.capacity, "Capacities must be equal"
        assert bloom_a.error == bloom_b.error, "Error rates must be equal"

        bloom_union = BloomFilter(bloom_a.capacity, bloom_a.error)
        bloom_union.data = bloom_a.data | bloom_b.data
        return bloom_union
```

如果我们插入的条目超过了我们对布隆过滤器所声明的容量，那么会发生什么呢？最终，布尔型列表中的所有条目都被设置成了 True，在这种情况下，我们说我们已看见了每一个条目。这意味着布隆过滤器对所设置的初始容量很敏感，如果我们正在处理一个不知道大小的数据集（例如，数据流），情况就会相当恶劣。

处理这种情况的一种方法就是使用布隆过滤器的变体，被称作可扩展的布隆过滤器[①]。它们通过一种特别的方法[②]把误差率不同的多个布隆过滤器串联在一起来工作。通过这样做，我们能够保证一个总体的误差率，并且当需要更多容量时，仅仅增加一个新的布隆过滤器。为了检测我们之前是否曾经看到过一个条目，我们只是在所有的子布隆过滤器上进行遍历，直到我们找到该对象或者消耗完列表。这种结构的一个简单实现可以在例 11-25 中见到，我们使用了之前的布隆过滤器作为底层机能，并且有一个计数器来简化了解什么时候该增加一个新的布隆过滤器。

例 11-25　简单的扩展布隆过滤器实现

```
from bloomfilter import BloomFilter

class ScalingBloomFilter(object):
    def __init__(self, capacity, error=0.005,
```

① Almeida, P. S., Baquero, C., Preguica, N., and Hutchison, D.“可扩展布隆过滤器”。信息处理信件 101: 255　261. doi:10.1016/j.ipl.2006.10.007。

② 错误值实际上会降低，类似于几何级数。这样，当你采用所有误差率的乘积时，它就趋近于所期望的误差率。

```python
                 max_fill=0.8, error_tightening_ratio=0.5):
        self.capacity = capacity
        self.base_error = error
        self.max_fill = max_fill
        self.items_until_scale = int(capacity * max_fill)
        self.error_tightening_ratio = error_tightening_ratio
        self.bloom_filters = []
        self.current_bloom = None
        self._add_bloom()

    def _add_bloom(self):
        new_error = self.base_error *
                    self.error_tightening_ratio ** len(self.bloom_filters)
        new_bloom = BloomFilter(self.capacity, new_error)
        self.bloom_filters.append(new_bloom)
        self.current_bloom = new_bloom
        return new_bloom

    def add(self, key):
        if key in self:
            return True
        self.current_bloom.add(key)
        self.items_until_scale -= 1
        if self.items_until_scale == 0:
            bloom_size = len(self.current_bloom)
            bloom_max_capacity = int(self.current_bloom.capacity * self.max_fill)

            # We may have been adding many duplicate values into the Bloom, so
            # we need to check if we actually need to scale or if we still have
            # space
            if bloom_size >= bloom_max_capacity:
                self._add_bloom()
                self.items_until_scale = bloom_max_capacity
            else:
                self.items_until_scale = int(bloom_max_capacity - bloom_size)
        return False

def __contains__(self, key):
        return any(key in bloom for bloom in self.bloom_filters)

def __len__(self):
        return sum(len(bloom) for bloom in self.bloom_filters)
```

处理这种情况的另一种方式就是使用一种称作计时布隆过滤器的方法。这种变体允许元素超时而被移出数据结构，这样就为更多的元素腾出了空间。对流处理尤其有效，因为我们能够让元素超时，比如说一小时之后，而且把容量设置得足够大以便能处理我们每小时看到的数据量。以这种方式使用布隆过滤器将会带给我们对最近一小时内发生的事情的良好视图。

使用这种数据结构感觉和使用一个集合对象很像。在下面的交互中，我们使用可扩展的布隆过滤器来增加几个对象，测试下是否曾经见到过它们，接着尝试以实验的方法找到假阳性率：

```
>>> bloom = BloomFilter(100)

>>> for i in xrange(50):
   ....:     bloom.add(str(i))
   ....:

>>> "20" in bloom
True

>>> "25" in bloom
True

>>> "51" in bloom
False

>>> num_false_positives = 0

>>> num_true_negatives = 0

>>> # None of the following numbers should be in the Bloom.
>>> # If one is found in the Bloom, it is a false positive.
>>> for i in xrange(51,10000):
   ....:     if str(i) in bloom:
   ....:         num_false_positives += 1
   ....:     else:
   ....:         num_true_negatives += 1
   ....:

>>> num_false_positives
54

>>> num_true_negatives
9895

>>> false_positive_rate = num_false_positives / float(10000 - 51)

>>> false_positive_rate
0.005427681173987335

>>> bloom.error
0.005
```

为了联合多个条目组，我们也可以用布隆过滤器做并集：

```
>>> bloom_a = BloomFilter(200)

>>> bloom_b = BloomFilter(200)

>>> for i in xrange(50):
   ...:     bloom_a.add(str(i))
   ...:

>>> for i in xrange(25,75):
   ...:     bloom_b.add(str(i))
   ...:

>>> bloom = BloomFilter.union(bloom_a, bloom_b)

>>> "51" in bloom_a # ❶
Out[9]: False

>>> "24" in bloom_b # ❷
Out[10]: False

>>> "55" in bloom # ❸
Out[11]: True

>>> "25" in bloom
Out[12]: True
```

❶ 值“51”不在 bloom_a 中。

❷ 同样，值“24”不在 bloom_b 中。

❸ 无论如何，bloom 对象包含了在 bloom_a 和 bloom_b 中的所有对象！

使用它的一个警告就是你只能对两个具有相同容量和误差率的布隆过滤器做并集。而且，最终的布隆过滤器所使用的容量能够与组成它的两个做并集的布隆过滤器所使用的容量之和一样大。这意味着你可能从两个布隆过滤器开始，它们各自填满了不到一半容量，当你把它们做并集联合起来时，就会得到一个新的超过容量并且不可靠的布隆过滤器！

11.6.4 LogLog 计数器

LogLog 类型的计数器基于散列函数的单个比特位也能被看作是随机的事实。那就是说，一个散列的第一个比特是 1 的概率是 50%，前两个比特是 01 的概率是 25%，而前三个比特是 001 的概率是是 12.5%。知道了这些概率，让散列在开始处保持最多的 0（例如，最不可能的散列值），我们就能估算出我们曾经见到了多少项条目。

一个对这种方法很好的类比就是投掷硬币。想象一下我们将要投掷一枚硬币 32 次，而且每次都是正面朝上。32 这个数字来源于我们使用 32 比特的散列函数的事实。如果我们投掷一次，它反面朝上，我们就存储数字 0，因为我们的最佳尝试接连产生了 0 次正面。既然我们知道在这枚硬币投掷背后的概率，我们也能够告诉你我们最长的序列就是 0 长度的，你能估算出我们尝试实验了 2^0 = 1 次。如果我们继续投掷硬币而且能够在得到一次反面之前得到 10 次正面，那么我们就能存储数字 10。使用相同的逻辑，你就能估算出我们尝试实验了 2^10 = 1024 次。使用这个系统，我们所能计数的最大值就是我们所考虑投掷的最大次数（对于 32 次投掷来说，这就是 2 ^ 32 = 4294967296）。

为了使用 LogLog 类型计数器对这个逻辑进行编码，我们对输入的散列值采用了二进制的表示，并且观察在我们看到第一个 1 之前有多少个 0。散列值能够被视作一个 32 次硬币投掷的序列，0 意味着投掷出了一次正面朝上，1 意味着投掷出了一次反面朝上（例如，000010101101 意味着我们在得到第一次反面之前投掷出了 4 次正面，010101101 意味着我们在第一次投掷出反面之前投掷出了一次正面）。这带给我们一个在得到这个散列值前尝试过几次的想法。在这个系统背后的数学问题几乎等价于 Morris 计数器，除了一个主要的例外：我们是通过查看实际的输入而不是使用一个随机数生成器来获得“随机”值的。这意味着如果我们继续给一个 LogLog 计数器增加相同的值，它的内部状态将不会改变。例 11-26 展示了一个 LogLog 计数器的简单实现。

例 11-26　LogLog 计数器的简单实现

```
import mmh3

def trailing_zeros(number):
    """
    Returns the index of the first bit set to 1 from the right side of a 32-bit
    integer
    >>> trailing_zeros(0)
    32
    >>> trailing_zeros(0b1000)
    3
    >>> trailing_zeros(0b10000000)
    7
    """
    if not number:
        return 32
    index = 0
    while (number >> index) & 1 == 0:
        index += 1
    return index
```

```
class LogLogRegister(object):
    counter = 0
    def add(self, item):
        item_hash = mmh3.hash(str(item))
        return self._add(item_hash)

    def _add(self, item_hash):
        bit_index = trailing_zeros(item_hash)
        if bit_index > self.counter:
            self.counter = bit_index

    def __len__(self):
        return 2**self.counter
```

这个方法的最大缺点就是我们可能得到一个一开始就增加计数器的散列值，这会歪曲我们的估算。这就类似于在第一次尝试时投掷出了 32 次反面。为了弥补，我们应该让许多人同时投掷硬币并且组合他们的结果。大数定律告诉我们当增加越来越多的投掷者时，总体统计量越少受到单独的投掷者的异常样本的影响。我们组合多个结果的确切方式就是 LogLog 类型方法的根本性差异（经典的 LogLog、SuperLogLog、HyperLogLog、HyperLogLog++等）。

我们能够完成这个“多个投掷者”方法，通过采用散列值的首对比特位来指定我们的投掷者中哪一位投掷者具有特定的结果。如果我们采用散列值的前 4 个比特，这意味着我们有 `2^4=16` 个投掷者。既然我们这次选择了使用前 4 个比特，我们只剩下了 28 个比特（对应于每一个硬币投掷者独立地投掷 28 次硬币），这意味着每一个计数器只能计数到 `2^28 = 268435456`。此外，有一个依赖于投掷者数量的常数（alpha）对估算做正则化[①]。所有这些一起带给我们一个具有 $1.05/\sqrt{(m)}$ 的准确度的算法，m 是所用寄存器的数量（或者说投掷者）。例 11-27 展示了一个简单的 LogLog 算法的实现。

例 11-27　简单的 LogLog 实现

```
from llregister import LLRegister
import mmh3

class LL(object):
    def __init__(self, p):
        self.p = p
        self.num_registers = 2**p
        self.registers = [LLRegister() for i in xrange(int(2**p))]
        self.alpha = 0.7213 / (1.0 + 1.079 / self.num_registers)

    def add(self, item):
        item_hash = mmh3.hash(str(item))
```

① 一个对基础 LogLog 和 SuperLogLog 算法的完整描述可以在 http://bit.ly/algo rithm_desc 中找到。

```
        register_index = item_hash & (self.num_registers - 1)
        register_hash = item_hash >> self.p
        self.registers[register_index]._add(register_hash)

    def __len__(self):
        register_sum = sum(h.counter for h in self.registers)
        return self.num_registers * self.alpha *
                                  2 ** (float(register_sum) / self.num_registers)
```

这个算法除了使用散列值作为一个指示器来把相似的条目去重之外，还有一个可调制的参数，能够用来在要达到哪种类型的准确度相对于你想要做出的空间妥协之间进行调拨。

在`__len__`方法中，我们对来自于所有单个 LogLog 寄存器的估算值求它们的平均值。无论如何，这不是我们组合数据的最高效的方式！这是因为我们可能得到一些不幸的散列值使得某个特定的寄存器有孤峰值，而其他的则还处于低值。正因为如此，我们只能达到 $O\left(\frac{1.30}{\sqrt{m}}\right)$ 的误差率，m 是所使用的寄存器的数量。

SuperLogLog[①]被设计用来修正这个问题。使用这个算法，只有最低的 70%的寄存器用来做大小估算，它们的值被一个由限制规则给出的最大值所束缚。这个附加特性把误差率减少到了 $O\left(\frac{1.05}{\sqrt{m}}\right)$。

我们通过忽略信息的方式来得到一个更好的估算值，这是违反直觉的！

最后，HyperLogLog[②]在 2007 年出现，并且给我们带来了进一步的准确度收益。它仅仅通过改变对单个寄存器做平均的方法就做到了：我们使用了球状平均方案，对结构可能所处的不同的边缘情况都做了特殊的考虑，而不是仅仅做平均值。这带给我们当前最好的误差率 $O\left(\frac{1.04}{\sqrt{m}}\right)$。此外，这个公式移除了对 SuperLogLog 来说是必须的排序操作。当你设法插入大量条目时，这能大大提升数据结构的性能。例 11-28 展示了 HyperLogLog 的一个简单实现。

例 11-28　简单的 HyperLogLog 实现

```
from ll import LL
import math
```

① Durand, M., and Flajolet, P. “大基数的 LogLog 计数”。会刊：ESA 2003, 2832 (2003): 605　617. doi:10.1007/978-3-540-39658-1_55。

② Flajolet, P., Fusy, E., Gandouet, O., et al. “HyperLogLog：对近似最优的基数估计算法的分析”。2007 年算法分析国际会议会刊，(2007)：127-146。

```
class HyperLogLog(LL):
    def __len__(self):
        indicator = sum(2**-m.counter for m in self.registers)
        E = self.alpha * (self.num_registers**2) / float(indicator)

        if E <= 5.0 / 2.0 * self.num_registers:
            V = sum(1 for m in self.registers if m.counter == 0)
            if V != 0:
                Estar = self.num_registers *
                        math.log(self.num_registers / (1.0 * V), 2)
            else:
                Estar = E
        else:
            if E <= 2**32 / 30.0:
                Estar = E
            else:
                Estar = -2**32 * math.log(1 - E / 2**32, 2)
        return Estar

if __name__ == "__main__":
    import mmh3
    hll = HyperLogLog(8)
    for i in xrange(100000):
        hll.add(mmh3.hash(str(i)))
    print len(hll)
```

HyperLogLog++是唯一的进一步增加了准确度的算法，当数据结构相对空时，增加了它的准确度。当更多的条蜜插入时，这个方案转向于标准的 HyperLogLog。其实这是相当有用的，因为 LogLog 类型计数器的统计量需要许多准确的数据——有一种允许使用更少量的条目来获取更高的准确度的方案大大提高了这种方法的可用性。这种额外的准确度通过一个更小的但是更准确的 HyperLogLog 结构来达到，它以后能够被转换成一个原先所请求的更大的结构。也有一些根据经验得出的常数被用在大小估算中来消除偏差。

11.6.5 真实世界的例子

为了更好理解数据结构，我们首先创建了一个具有很多唯一键的数据集，接着创建一个具有重复条目的数据集。当我们把这些键输入这些数据结构后，只是观察它们并周期性地询问：“已经有多少唯一条目了？”，图 11-5 和图 11-6 展示了结果。我们能看到包含更多有状态的变量（例如 HyperLogLog 和 KMinValues）的数据结构做得更好，因为它们更健壮地处理了糟糕的统计量。另一方面，如果产生了一个不幸随机数或者不幸散列值，Morris 计数器和单个 LogLog 寄存器就可能快速发生很高的误差率。无论如何，对于大多数算法而言，我们知道有状态变量的数量直接和所能确保的错误相关，所以这就是有意义的。

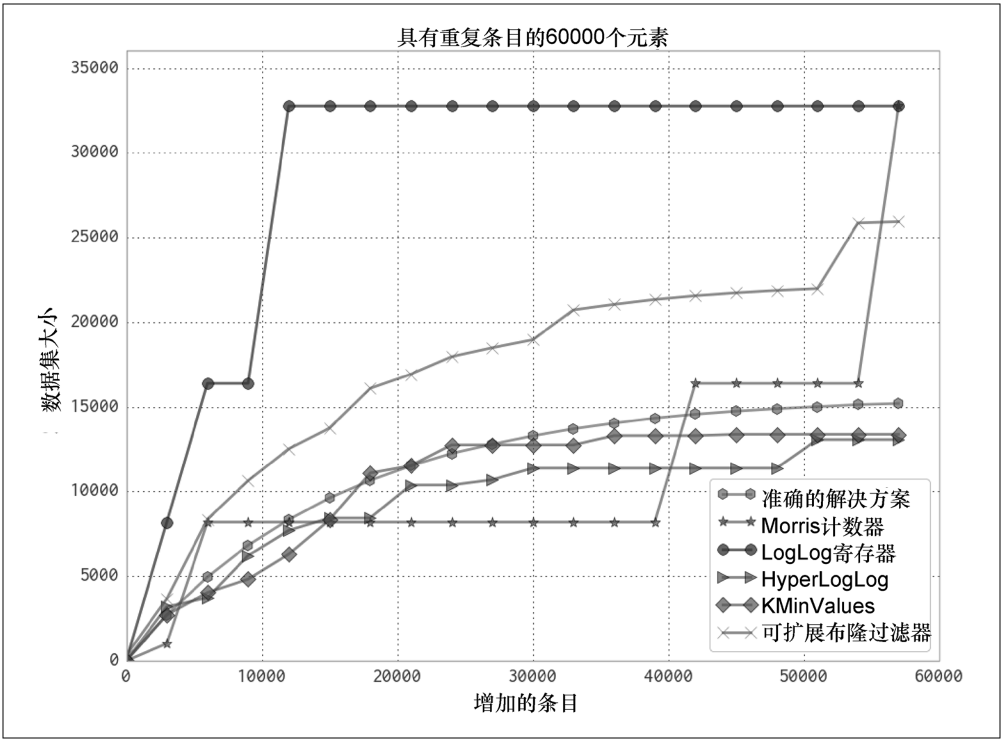


图 11-5　各种不同的概率数据结构对于重复数据的对比

只看具有最佳性能（实际上，你可能会使用的那些）的概率数据结构，我们可以总结出它们的效用和大概的内存使用（参见表 11-2）。我们可以看到内存使用的巨大变化取决于我们所关心的问题。这仅仅强调了这样一个事实：当使用一个概率数据结构时，你必须首先考虑在继续进行之前，有关数据集的什么问题是你真正需要回答的。也要注意只有布隆过滤器的大小取决于元素的数量。HyperLogLog 和 KMinValue 的大小只取决于误差率。

作为另一个更为真实的测试，我们选择使用了一个派生自维基百科文本的数据集。我们运行了一个很简单的脚本来从所有的文章中抽取出具有 5 个字符的所有单词符号，并把它们存储在一个用换行符分隔的文件中。那么问题就是："有多少唯一的符号？"在表 11-3 中能看到结果。此外，我们企图使用来自于 11.4 节中的 datrie 树来回答相同的问题（这个 trie 树被选来与其他数据结构做对比，因为它提供了良好的压缩性，然而却还是足够鲁棒来处理完整的数据集）。

来自于这个实验的主要副品就是如果你能够特殊化你的代码，你就能得到令人惊叹

的速度和内存收益。贯穿于整本书，这都是真实的：当你对 6.4.2 节中的代码进行特殊化处理时，我们就同样能得到速度提升。

表 11-2　　主要概率数据结构的比较

	误差率	并集[a]	交集	是否包含	大小[b]
HyerLogLog	是（$O\left(\frac{1.04}{\sqrt{m}}\right)$）	是	不[c]	不	2.704MB
KMinValues	是（$O\left(\frac{\sqrt{2}}{m(m-2)}\right)$）	是	是	不	20.372MB
布隆过滤器	是（$O\left(\frac{0.78}{\sqrt{m}}\right)$）	是	不[c]	是	192.8MB

a．并集操作不会增加误差率。

b．数据结构大小，具有 0.05%的误差率，100000000 个唯一元素，使用一个 64 比特的散列函数。

c．这些操作能够做，但是对准确度有一个不可忽视的惩罚。

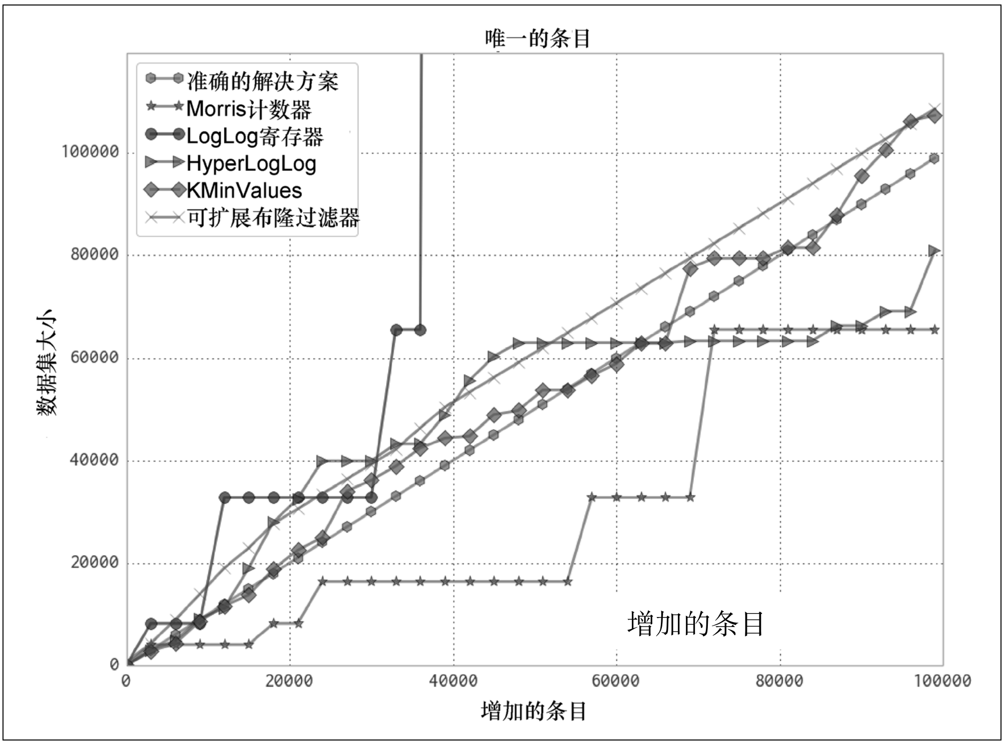


图 11-6　各种不同的概率数据结构对于唯一数据的对比

概率数据结构是一种对你的代码做特殊化处理的算法。我们只存储所需的数据来以给定的错误上界回答特定的问题。通过只处理所给信息的子集，我们不仅能够让内存占用空间减少很多，而且也能够在这些结构上更快地执行大部分操作（就如在表 11-3 中所能见到的那样，插入 datrie 树的时间比任何一种概率数据结构要来得多）。

表 11-3　　对维基百科中的唯一单词的数量的估计

	元素	相对误差	处理时间 [a]	结构大小 [b]
Morris 计数器 [c]	1 073 741 824	6.52%	751 秒	5bit
LogLog 寄存器	1 048 576	78.84%	1 690 秒	5bit
LogLog	4 522 232	8.76%	2 112 秒	5bit
HyperLogLog	4 983 171	-0.54%	2 907 秒	40KB
KMinValues	4 912 818	0.88%	3 503 秒	256KB
可扩展布隆过滤器	4 949 358	0.14%	10 392 秒	11509KB
datrie 树	4 505 514d	0.00%	14 620 秒	114068KB
真值	4 956 262	0.00%	-----	49558KB [e]

a．处理时间已经被调整过，移除了从磁盘读取数据集的时间。我们也使用之前所提供的简单实现来做测试。

b．结构大小是在给出数据量前提下的理论值，因为所用的实现没有去优化。

c．因为 Morris 计数器不会对输入去重，结构体大小和相对误差按照值的总数量给出。

d．因为一些编码问题，datrie 树不能加载所有的键。

e．只考虑唯一的符号，数据集是 49558KB，考虑所有的符号，则是 8.742GB。

作为结果，无论你是否使用概率数据结构，你应该总是记住你打算要问你的数据哪些问题，以及你如何能够最高效地存储数据以便去问那些特别的问题。这可能归结于使用一种特定类型的列表，使用一种特定类型的数据库索引，或者甚至可能使用一个概率数据结构来抛弃掉除了相关数据以外的所有数据！

第 12 章
现场教训

读完本章之后你将能够回答下列问题

- 成功的创业公司如何处理大量的数据和机器学习？
- 什么样的监控和部署技术让系统保持稳定？
- 成功的 CTO 学到了关于技术和团队的什么教训？
- PyPy 怎样被广泛部署？

在本章中，我们从成功的公司那收集了一些故事，在这些公司里 Python 被用在了大数据量和速度关键的场景下。这些故事由每个组织中具有多年经验的关键人物写成。他们不仅共享了技术选择，而且也分享了一些经过努力获得的来之不易的智慧。

12.1 自适应实验室（Adaptive Lab）的社交媒体分析（SoMA）

Ben Jackson (adaptivelab.com)

自适应实验室是一家座落于伦敦的技术城市（Tech City）区——肖尔迪奇区的产品开发和创新公司。我们应用了我们的精益生产、以用户为中心的产品设计和交付方法，与包括从创业公司到大企业在内的许多公司展开了广泛的合作。

YouGov 是一家全球市场研究公司，它所声称的雄心抱负就是提供一个连续的、准确的数据流来洞见全世界的人们在想什么和做什么——那就是我们要设法提供给它们的东西。自适应实验室设计了一种方式来被动倾听发生在社交媒体上的真实的

讨论，并且洞见了用户对定制范围内的一些主题上的情感。我们构建了一个可扩展的系统，能够无穷无尽地捕获大量的信息流，处理并存储它们，还能够通过一个强大的可过滤的接口来实时展现它们。系统是使用 Python 来构建的。

12.1.1 自适应实验室（Adaptive Lab）使用的 Python

Python 是我们的核心技术之一。我们在性能关键的应用中使用它，并且无论何时，我们和具有公司内部使用 Python 技术的客户一起工作，这样我们为客户所做的产品能够在客户公司内部被采用。

Python 对于小型的、自包含的并且长期运行的守护进程来说是理想的，它很好用，具有类似 Django 和 Pyramid 那样灵活而特性丰富的 Web 框架。Python 社区是繁荣的，这意味着那里有大量的开源工具库来允许我们快速而充满信心地去构建，让我们集中精力于新型的、创新性的东西来为客户解决问题。

贯穿于我们整个项目的是，我们在自适应实验室复用了用 Python 构建的几个工具，但是却能够以一种语言无关的方式来使用。例如，我们使用了 SaltStack 来做服务器配给，使用了 Mozilla 的 Circus 来管理长期运行的进程。当一个工具是开源的并且以我们所熟悉的语言来写成时，所带给我们的好处就是如果我们发现任何问题，我们能够自己解决，并且把那些解决方案提上去，这让社区也得到收益。

12.1.2 SoMA 的设计

我们的社交媒体分析工具需要处理高吞吐量的社交媒体数据，以及存储和获取大量的实时信息。在研究了各种各样的数据存储和搜索引擎后，我们决定把 Elasticsearch 作为我们的实时文档存储。就如它的名字所示的那样，它是高可扩展的，但是也非常容易使用，而且能够提供统计和搜索的应答——对我们的应用来说是理想的。Elasticsearch 本身是用 Java 构建的，但是就像任何一个现代系统中的架构良好的组件那样，它有良好的 API，而且使用 Python 库和教程良好地为它服务。

我们设计的系统使用了在 Redis 中所持有的 Celery 作为队列来快速分发大量的数据流给任意数量的服务器来做独立处理和索引。整个复杂系统的每一个组件被设计得小型化、独立而简单，并且能够相互隔离地来工作。每一个组件集中于一个任务上，比如分析一个对话的感情色彩或者准备一个文档来索引编入 Elasticsearch。一些组件被配置成使用 Mozillar 的 Circus 以守护进程的方式来运行，让所有的进程保持运行并允许它们在单独的服务器上水平扩张或收缩。

SaltStack 被用来定义和配给复杂的集群，并处理所有的库、语言、数据库和文档存储的设置。我们也使用 Fabric，一个在命令行上运行任意任务的 Python 工具。在

代码中定义服务器有许多益处：与生产环境的完全对等，配置的版本控制，把所有东西放在了一起。它也为配置和集群所需依赖项的文档化发挥了作用。

12.1.3 我们的开发方法论

我们意图让项目的新手尽可能容易地能够快速而有信心地着手添加代码和部署。我们使用 Vagrant 来在本地构建复杂系统，在虚拟机内部完全和在生产环境中等价。一个简单的vagrant up就是一个新手所需的全部命令来着手设置他们工作所需的所有依赖。

我们以敏捷方式工作，一起计划、讨论架构方面的决定，并且在任务估计上达成了一致。对于 SoMA 来说，我们决定在每一次迭代中要包括一些被视作修正“技术债”的任务。也有些对系统进行文档化的任务包括了进来（我们最终建设了一个维基来掌管这个日益膨胀的系统的所有知识）。在每个任务结束后，组员相互检查代码来做完整性检查，提供反馈并理解将要加入系统中的新代码。

一个良好的测试套件有助于提升信心，任何改动将不会造成已经存在的特性失效。集成测试在一个类似 SoMA 的系统中至关重要，由许多移动的部件组成。一个脚手架的环境提供了一种测试新代码性能的方法。尤其是在 SoMA 上，只有通过在产品中所见到的那样大型的数据集上做测试，才能让问题出现并得到处理，所以经常需要在独立的环境下重现那样的数据量。亚马逊的弹性计算云（EC2）带给我们着手此事的灵活性。

12.1.4 维护 SoMA

SoMA 系统连续运行着，并且它消费的信息量与日俱增。我们不得不对数据流中的峰值、网络问题以及它所依赖的任何第三方服务提供者中的问题做出解释。因此，为了让我们自己的事情变得简单，SoMA 被设计成自我修复，无论何时只要它有能力就自我修复。多亏了 Circus，崩溃的进程将恢复运行并从它们断线的地方开始继续运行任务。一个任务将在队列中缓存起来，直到进程能够消费它，并且当系统恢复时，有足够的缓存空间来堆积任务。

我们使用 Server Density 来监控许多 SoMA 服务器。它设置起来很简单，但是相当强大。当一有问题可能要发生时，一个指派的工程师就能够在手机上接收到推送消息，这样他可以及时做出反应来确保它不要变成一个真正的问题。使用 Server Density，用 Python 来写定制插件也非常容易，例如，来设置 Elasticsearch 表现方面的快速报警。

12.1.5 对工程师同行的建议

最主要的是，你和你的团体需要有信心并感到轻松愉快——将要部署到现场环境中

去的项目是会无故障运行的。为了得到这一点，你必须要回溯工作，花费时间在系统的所有组件上来给你那种舒服的感觉。让部署简单并且安全可靠，使用一个脚手架环境来测试在真实世界的数据下的性能，确保你具有一个良好可靠的高覆盖率的测试套件，实现一个过程把新代码整合进系统中，确保技术债尽早地得到解决。你越是支撑强化你的技术架构并且提高你的过程，你的团队就会越来越愉快和成功地来把合适的解决方案工程化。

如果缺少扎实的代码基础和生态系统，但是商业上却强迫你把活干起来，那只会产生有问题的软件。那将是你的责任来推动并争取时间对代码、测试以及需要推向落地的运营作的操作做出增量式的改进。

12.2 使用 RadimRehurek.com 让深度学习飞翔

Radim Řehůřek (radimrehurek.com)

当 Ian 请我来为本书写写我在 Python 和优化方面的“来自现场的教训”时，我立即思考，“告诉他们你怎样制作了一个比 Google 的 C 语言原始版本要更快的 Python 移植版！”。Google 的深度学习的典型代表比原始的 Python 实现快了 12000 倍，这是一个创造机器学习算法的鼓舞人心的故事。任何人可以写出糟糕的代码，接着再鼓吹巨大的速度提升。但是有点令人吃惊的是，优化过的 Python 移植版也运行得比 Google 团队所写的原始代码几乎快 4 倍！那就是，比晦涩的、配置紧凑的优化过的 C 代码快 4 倍。

但是在吸取“机器层面”的优化教训之前，有一些通用的关于“人的层面”的优化建议。

12.2.1 最佳时机

我运转着一个小型的专注于机器学习的咨询业务，我的同事和我帮助公司让数据分析的混乱世界变得有意义，目的是赚钱或节省成本（或两者兼得）。我们帮助客户为数据处理设计和构建令人惊叹的系统，尤其是在文本数据方面。

客户范围从大型的跨国企业到新生的初创公司，尽管每个项目各不相同并且需要不同的技术栈来插入客户已经存在的数据流和管道中，Python 是明显的优先选择。不是要白费口舌，Python 的简单直接的开发哲学、可塑性，以及丰富的库生态系统让它成为了一个理想的选择。

首先，一些“来自现场”的关于是什么发挥着作用的思考：

- **沟通，沟通，沟通**。这很明显，但值得重复提及。在决定一种方法前，在更高的层面（业务）上理解客户的问题。坐下来谈谈他们认为所需要的东西（基于在联系你之前他们对什么是有可能做的以及/或者他们从谷歌上搜索到的片面的知识），直到对他们真正所需的东西一清二楚，而免于晦涩和偏见。要事先对验证解决方案的方式达成一致。我想要把这个过程可视化成构筑一条绵长而曲折的道路：得到正确的开端（问题定义，可利用的数据资源）和正确的终点（评估，解决方案的优先级），以及属于这两者之间的路径。

- **寻找有前途的技术**。一个得到良好理解的、健壮的、正在得到关注的，然而在工业领域还是相对模糊的新兴技术能够给客户（或你自己）带来巨大的价值。例如，几年前，Elasticsearch 是一个罕为人知的有一些粗糙的开源项目。但是我评估了它的方法，认为它是扎实的（构建于 Apache Lucene 之上，提供了副本、集群分片等）并把它的用途推荐给了客户。结果我们使用 Elasticsearch 作为核心构建了一个搜索系统，相比可观的替代方案（大型的商业数据库），帮客户省下了在版权、开发和维护方面的巨额资金。甚至更重要的就是，使用一个崭新的、灵活又强大的技术给产品带来了巨大的竞争优势。现在，Elasticsearch 已经进入了企业市场并传递出无与伦比的竞争优势——每个人都知道和使用它。掌握正确的时机就是我所能声称的“最佳时机”，让价值/成本之比达到最大化。

- **KISS（让它简单，愚蠢！）**这是另一种不必花脑筋的事。最好的代码就是你不必编写和维护的代码。从简单开始，并且在必须的地方改进和迭代。我倾向于遵循 UNIX 的“做一件事情，并把它做好”哲学的工具。宏伟的编程框架可能是吸引人的，让每样可设想到的事情处于同一屋檐下并且干净地适配在一起。但是你迟早总是需要一些巨大的框架所设想不到的东西，然后甚至看起来简单（从概念上）的改动串联起了一场噩梦（从编程角度而言）。宏伟的项目以及它们所包含的全部 APIs 在它们自身的重量下趋向于崩溃。要使用模块化、专注的、尽可能短小而简单的工具。要倾向于对简单的可视化检查开放的文本格式，不然除非是性能上的强行规定。

- **在数据管道中使用手动的完整性检查**。当优化数据处理系统时，容易停留在“二进制的思维”模式中，使用紧凑的管道、高效的二进制数据格式和压缩过的 I/O。当数据以不可见和未经检查的（可能除了它的类型）方式通过系统时，它保持着不可见性，直到某些情况彻底爆发。然后调试开始。我建议把一些简单的日志消息撒遍整个代码，把在各种不同的内部处理点上展示数据的形态看作一个良好的实践——没有什么花哨的，只是模拟 UNIX 的 head 命令，挑选并对一

些数据点做可视化。这样不仅有助于前面提到过的调试，而且以人类可读的格式看到数据经常会令人惊讶地产生“啊!”的时刻，甚至于在一切看起来都良好的情况下。奇怪的符号化！它们承诺的输出总是以 latin1 来编码！这种语言的文档怎么会出现在那里？图像文件泄漏到了期待和解析文本文件的管道中！这些常常超越了由自动类型检查或者固定的单元测试所提供的认识力，暗示着跨越组件边界的问题。真实世界的数据是混乱的。早期捕获到事件甚至不一定会产生异常或者明显的错误。宁可失之于过度冗长。

- **小心地在潮流中导航**。只是因为一个客户一直听说 X 并说出他们必须也要 X 并不意味着他们真正需要它。它可能是一个市场问题而不是一个技术问题，所以要仔细分辨并相应地传达。X 随时间变化，因为炒作的浪潮来来去去，一个最近的值将会是 X＝ 大数据。

总之，谈论业务足够了——这就是我如何用 Python 获得了比 C 运行更快的 word2vec。

12.2.2 优化方面的教训

word2vec 是一个允许检测相似单词和短语的机器学习算法。随着文本分析和搜索引擎优化（SEO）方面的有趣应用以及附于其上的谷歌的光辉品牌，初创公司和业务蜂拥而上地利用起这个新工具。

不幸的是，唯一可利用的代码就是由谷歌自己所生产的，一个用 C 语言编写的开源 Linux 命令行工具。这个工具得到了良好的优化，但是相当难以用来实现。我决定要把 word2vec 移植到 Python 上的主要理由就是我能够把 word2vec 扩展到其他的平台，让它更易于为客户集成和扩展。

在这里不关乎细节，但是 word2vec 需要一个具有很多输入数据的训练阶段来产生有用的相似模型。例如，谷歌的工程师们在他们的 GoogleNews 数据集上运行 word2vec，训练了大约 1000 亿个单词。这种尺度的数据集明显不适合放入 RAM 中，所以必须采用一个节约内存的方法。

我已创造了一个机器学习库 gensim，目标准确定位于这种内存优化的问题：数据集不再是小规模（“小规模”就是任何能完全放入 RAM 中的事物)，而又没足够大到有必要使用 PB 字节规模的 MapReduce 计算机集群的地步。令人惊奇的是，这个“千兆”范围的问题适合于真实世界的一大部分情况，包括 word2vec。

细节在我的博客上有描述，但是这里有一些外带的优化：

- **流化你的数据，观察你的内存**。让你的输入按照一次一个数据点的方式被访问和处理，目的是得到一个小而固定的内存占用空间。流化的数据点（在 word2vec 的情况下是句子）可能为了性能在内部被组织成更大的批量（例如同一时刻处理 100 条句子），但是高级的、流化的 API 证实了一种强大而又灵活的抽象。Python 语言使用它内置的产生器，很自然和优雅地支持这种模式—— 一场真正优美的问题和技术的竞赛。避免致力于那些把一切都装载进 RAM 中的算法和工具，除非你知道数据总是保持小规模，或者你不介意以后自己去重新实现一个生产版本。

- **利用 Python 的丰富生态系统**。我从一个可读的、使用 numpy 的干净的 word2vec 的移植版开始。Numpy 在本书的第 6 章中被深度地涉猎到了，但是作为一个短小的提醒，这是一个美妙的库，是 Python 科学社区的基石，是用 Python 做数字爆破的事实标准。挖掘 numpy 的强大数组接口、内存访问模式以及为超速的通用矢量操作所包装的 BLAS 例程产生了简洁、干净和快速的代码——比原生的 Python 代码要快好几百倍。通常我在这点上就此打住，但是“几百倍更快”还是比谷歌优化过的 C 版本要慢 20 倍，所以我要强调一下。

- **配置和编译热点**。word2vect 是一个典型的高性能计算应用，因为在一个内循环中的少数几行代码占用了 90%的整体训练运行时间。在这里，我用 C 重写了一个单独的核心例程(大约 20 行代码),使用一个外部的 Python 库,把 Cython 作为胶水。尽管技术上光彩夺目，我却不认为 Cython 从概念上是一个特别方便的工具——它基本上就像在学习一门其他语言，一种在 Python、numpy 和 C 之间的非直觉的混合物,而有它自己的说明和特质。但是直到 Python 的 JIT（即时编译器）技术成熟前，Cython 可能就是我们的最佳赌注。使用一个 Cython 编译成的热点，word2vec 的 Python 移植版本的性能现在与原来的 C 代码不相上下。从一个干净的 numpy 版本开始的另一个优势就是通过与更慢但是正确的版本做对比，我们就得到了免费的正确性测试。

- **知道你的 BLAS**。numpy 的一个干净特性就是它内部在可利用的地方包装了 BLAS（基础线性代数子例程）。这些是低级的例程集合，直接通过处理器供应商（英特尔、AMD 等）使用汇编、Fortran 或者 C 来做优化，被设计用于从一种特定的处理器架构中挤榨出最佳的性能。例如，调用一个 axpy 的 BLAS 例程来计算 `vector_y += scalar * vector_x`，这样比通用的编译器为一个等价的显式的循环所产生的代码要更快。把 word2vec 的训练表示成 BLAS 操作导致了额外的 4 倍速度提升，胜过了 C 版本的 word2vec 的性能。获胜了！公平来说，C 代码也能够链接 BLAS，所以这不是 Python 与生俱来的优势。numpy 只是让诸如此类的事物突显出来并让它们变得容易利用。

- **并行化和多核**。gensim 包含了一些算法的分布式集群实现。对于 word2vec 来说，我选取了在一台机器上的多线程方式，因为它的训练算法具有细粒度的本质。使用多线程也允许我们避免 Python 多进程所带来的 fork-without-exec 的 POSIX 问题，尤其是在与某些 BLAS 库混用的时候。因为我们的核心例程已经使用了 Cython，我们能够担负起释放 Python 的 GIL（全局解释器锁，请看 7.8 节中的“在一台机器上使用 OpenMP 来做并行解决方案”），这通常使得多线程对于 CPU 密集型任务无效。速度提升：在一台机器上使用 4 核另外又提高了 3 倍。

- **静态内存分配**。在这点上，我们每秒处理好几万条句子。训练是如此快速，甚至于没有什么类似于创建一个新的 `numpy` 数组（对每一个句子流调用 malloc）那样拖慢我们的速度。解决方案：预分配静态的“工作”内存并以良好的老 Fortran 的风格来周转。让我流泪了。在这里的教训就是尽量在干净的 Python 代码中保持记账和应用逻辑，并且要让优化过的热点保持精简。

- **具体问题的优化**。原来的 C 语言实现包含了具体的微观优化，例如在特定的内存边界对齐数组或者在内存的查找表中预先计算某些函数。一阵来自过去的令人怀念的风气，随着如今复杂的 CPU 指令流水线、内存缓存层级以及协处理器，这种优化已不再是确定的赢家。细心的剖析暗示着一定百分比的提高，可能不值得为之付出额外的代码复杂性。另外，使用注解和剖析工具来高亮出优化不够的点。使用你的领域知识来引入以准确度换取性能（或反之）的渐进算法。但是从不要把它当作信条，剖析倾向于使用真实的生产数据。

12.2.3 总结

在合适的地方优化。以我的经验来看，从来没有充分的沟通来完全确认问题范围、优先级以及和客户业务目标的关系——即“人的层面”上的优化。确认你交付了相关的问题，而不是为它迷失在“极客工具”中。当你卷起袖子准备干活时，让它值得去做！

12.3 在 Lyst.com 的大规模产品化的机器学习

Sebastjan Trepca (lyst.com)

Lyst.com 是一个位于伦敦的时尚推荐引擎，每个月有超过 2 000 000 个用户通过 Lyst 的艰苦抓取、清理和建模过程来学习到新时尚。它成立于 2010 年，得到了 2 000 万美元的融资。

Sebastjan Trepca 是技术创立者和 CTO，他使用 Django 创建了网站，Python 已经帮助了团队来快速地测试新的想法。

12.3.1 Python 在 Lyst 的地位

自从网站创建以来，Python 和 Django 就已经是 Lyst 的核心了。随着内部的项目成长，一些 Python 组件被其他工具和语言取代来适应系统不断成熟的需求。

12.3.2 集群设计

集群运行于亚马逊的 EC2 上。总共有大约 100 台机器，包括更新的 C3 实例，具有良好的 CPU 性能。

Redis 和 PyRes 一起作为队列来使用并存储元数据。主要数据格式是 JSON，目的是让人易于理解。`Supervisord` 让进程保持活跃。

Elasticsearch 和 PyES 被用来索引所有的产品。Elasticsearch 集群跨越 7 台机器存储了 6 千万个文档。Solr 被调研过，但是因为缺少实时的更新特性而评价不高。

12.3.3 在快速前进的初创公司中做代码评估

写出能够被快速实现的代码更佳，这样一个商业上的想法就能够被测试了，而不是花费大量时间企图在第一遍就写出“完美的代码”。如果代码是有用的，那么它就能被重构；如果在代码背后的想法是糟糕的，那么删除并移除一个特性是代价低廉的。这可能导致一个复杂的基础代码，有许多对象传来传去，但是只要团队花费时间去重构对业务有用的代码，这就是可接受的。

文档字串（docstring）在 Lyst 中使用很多——尝试过一个外部的 Sphinx 文档系统但是放弃了，仅仅是为了方便阅读代码。一个维基被用来对过程和更大的系统做文档化。我们也开始创建很小的服务，而不是把一切都塞进一份基础代码中。

12.3.4 构建推荐引擎

首先推荐引擎用 Python 来编码，使用 `numpy` 和 `scipy` 来计算。接下来，推荐引擎的性能关键部分使用 Cython 来加速。核心的矩阵分解运算完全用 Cython 来写，产生了一个数量级的速度提升。这主要归因于有能力写出超过 Python 中的 `numpy` 数组的高效循环，当矢量化时，有些东西用纯粹的 Python 特别慢，性能糟糕，因为它需要 `numpy` 数组的内存拷贝。罪魁祸首就是 `numpy` 中的复杂索引，总是要对被切片的数组创建一份数据拷贝：如果数据拷贝不是必要的或者故意要做的，Cython 的循环将会快得多。

随着时间的推移，系统的在线组件（负责在请求时执行推荐计算）被集成进了我们的搜索组件 Elasticsearch 中。在这个过程中，它们被翻译成了 Java 来允许与 Elasticsearch 做集成。这背后的主要原因不是因为性能，而是因为要让推荐引擎与一个完整而强大的搜索引擎做集成的用途，以允许我们更容易地把业务规则应用于所服务的推荐上来。Java 组件本身就特别简单，并实现了主要的高效稀疏矢量内积。更复杂的离线组件还是用 Python 来写，使用了 Python 科学栈（主要是 Python 和 Cython）的标准组件。

根据我们的经验，Python 作为一种原型语言更有用：类似 `numpy`、Cython 和 `weave`（更近的有 Numba）这样可利用的工具允许我们在代码中的性能关键部分取得十分良好的性能，然而却还是保留了 Python 的干净和强大的表达力，而低级的优化将会事与愿违。

12.3.5　报告和监控

Graphite 被用来做报告。当前，性能退化在部署之后能够用肉眼看到。这样就容易深入探查详细的事件报告或者缩小来看到站点行为的高层次报告并添加或者移除必要的事件。

一个为性能测试所做的更大的基础设施正在做内部设计。它将包括代表性的数据和用户场景来适当地测试新构建的站点。

一个作为脚手架的站点也会被用来让一小部分真正的访问者看到部署的最新版本——如果看到一个错误或者性能退化，那么它只会影响一小部分访问者，这个版本能够快速地回退。这将使得有错误的部署明显地降低成本并减少问题。

Sentry 被用来记录和诊断 Python 的栈跟踪信息。

Jekins 被用来和内存数据库配置做连续集成。这就能够做并行化的测试来让签入快速地暴露任何错误给开发者。

12.3.6　一些建议

有良好的工具来跟踪你所构建的东西的效能是真正重要的，并且在一开始是超级实用的。初创公司一直在变化并且工程不断地在演化：你从一个强烈探索性的阶段开始，用所有的时间构建原型并删除代码，直到你命中了金矿，然后你开始向更深处前进，提高代码和性能等。直到那时，一切都和快速迭代以及良好的监控/分析有关。我猜测这是重复了一遍又一遍的相当标准的建议，但是我想很多人没有真正体会到它是多么重要。

我不认为如今技术有多大影响，所以使用任何能为你工作的东西。然而在转移到类似 AppEngine 或者 Heroku 之类的寄宿环境之前，我要三思而行。

12.4 在 Smesh 的大规模社交媒体分析

Alex Kelly (sme.sh)

在 Smesh，我们生产的软件所摄取的数据来自于遍及 Web 的多种多样的 API，过滤、处理并聚合它们，然后使用数据来为各种客户构建定制的应用。例如，我们提供了技术在 Beamly 的双屏 TV 应用中推进了推特消息（tweet）的过滤和流化，为移动网络 EE 运行了一个品牌和营销监控平台，以及为谷歌运行了一堆 Adwords 数据分析项目。

为了做到那样，我们运行了各种各样的流和轮询服务，经常性地轮询推特（Twitter）、Facebook、YouTube 和许多其他内容服务，并每天处理几百万条推特消息。

12.4.1 Python 在 Smesh 中的角色

我们广泛地使用了 Python——我们的主要平台和服务使用它来构建。多种不同的可利用的库、工具和框架允许我们为所做的大多数事情来全盘使用它。

这种多样性带给我们能力来（有希望）为工作挑选出合适的工具。例如，我们已经创建了使用每一个 Django、Flask 和 Pyramid 的应用。每一个都有它的好处，我们能够为手头的任务挑选出合适的一个。我们为多任务使用 Celery，为和 AWS 交互使用 Boto，并为我们数据所需的一切使用 PyMongo、MongoEngine、redis-py、Psycopg 等。这个列表将不断延伸下去。

12.4.2 平台

我们的主要平台由一个中心 Python 模块组成，为数据输入、过滤、聚合和处理提供了钩子，还有多种其他的核心函数。项目的具体代码从那个核心中导入功能，然后根据每个应用的需求实现更多具体的数据处理和展现逻辑。

直到现在平台为我们工作良好，并允许我们构建相对复杂的应用来摄取和处理来自多种多样不同源的数据，而没有较多的重复的工作量。无论如何，它不是没有缺点——每个应用依赖于一个公共的核心模块，使得更新代码的过程位于那个模块中，并且让所有使用它的应用保持更新成了一项主要任务。

当前我们工作于一个项目上来重新设计核心软件并且前进到一个更加面向服务的架构（SoA）的方法中去。当平台成长时，寻找合适的时机来做出那种架构变化似

乎是大多数软件团队所面临的一项挑战。把组件构建成单独的服务具有开销，并且所需要的用来构建每个服务的深入的领域特定知识经常只有通过一个初始的迭代开发才能掌握，架构的开销对于解决手头的真正问题是一个阻碍。有希望的是，我们已经选择了一个明智的时机来重新审视我们的架构抉择从而继续前进。交给时间来说话。

12.4.3 高性能的实时字符串匹配

我们从推特的流 API 中消费了许多数据。当我们在推特消息中流动时，我们把输入字符串与一组关键字做匹配，这样我们就知道我们正在跟踪的每条推特消息与哪个词项有关。这不是那种具有低速率的输入或者小关键字集的问题，而是为每秒几百条推特消息与几百个或几千个可能的关键字做匹配，从而问题开始变得棘手。

我们不仅对关键字字符串是否存在于推特消息中感兴趣，而且对更复杂的模式与单词的边界，行的起始和结束以及可选用的作为字符串前缀的 # 和 @ 字符是否匹配感兴趣，这使得问题变得更加棘手。最有效的封装匹配知识的方法就是使用正则表达式。无论如何，在每秒几百条推特消息上运行几千个正则表达式是计算密集型的。之前，我们不得不在集群机器上运行许多工作节点来实时可靠地执行匹配。

知道这是系统中的主要性能瓶颈之后，我们尝试了各种不同方法来提高我们匹配系统的性能：简化正则表达式，运行足够的进程来确保我们充分利用了服务器的所有核，确保我们的所有正则表达式被编译过了，得到了合适的缓存，以及在 PyPy 下运行匹配任务，而不是在 CPython 下等。这些中的每一个做法都提升了一点点性能，但是要明白这种方式只是减少了我们的一小部分处理时间。我们正寻找一个数量级的速度提升，而不是一小部分改进。

已经很明显了，比起尝试提高每一次匹配的性能，我们需要在模式匹配发生前，降低问题空间大小。这样我们需要减少要处理的推特消息的数量，或者减少我们所需要去匹配推特消息的正则表达式的数量。抛弃到来的推特消息不是一个选项——那是我们感兴趣的数据。所以，我们想方设法减少我们所需的的模式数量来与到来的推特消息做比较，从而执行匹配。

我们开始看看各种不同的 trie 树结构来允许我们更高效地在多组字符串之间做模式匹配，并且遇到了 Aho-Corasick 字符串匹配算法。它被证明对于我们的使用场景是理想的。构建 trie 树的字典必须是静态的—— 一旦自动化结束，你不能给 trie 树添加新成员——但对我们来说，这不是问题，因为关键字集合在来自推特的一个会话流的持续时间里是静态的。当我们改变正在跟踪的词项时，我们必须从 API 断开并重新连接上 API，这样我们就能同时重建 Aho-Corasick trie 树。

使用 Aho-Corasick 来对照字符串处理输入同时找到所有可能的匹配，在一个时刻每步向前经过输入字串的一个字符，并且沿着 trie 树向下在下一层找到匹配的节点（或者没有找到，就如在案例中可能的那样）。这样，我们就能很快找到我们的哪一个关键词项可能存在于推特消息中。我们还是无法确切知道，因为 Aho-Corasick 中纯粹的字串与字串间的匹配不允许应用任何封装在正则表达式中的更复杂的逻辑，但是我们能够把 Aho-Corasick 匹配作为一个预过滤器来使用。不在字符串中的关键字无法匹配，这样基于出现在文本中的关键字，我们就知道仅仅只须尝试我们所有的正则表达式中的一个小小的子集。我们排除了绝大部分正则表达式并且对每条推特消息只需要处理极少数表达式，而不是对照每条输入评估成百上千个正则表达式。

通过把我们试图为每条推特消息来做匹配的模式数量降低到极少值，我们已经设法达成了我们所寻求的速度提升。依赖于 trie 树的复杂性以及输入推特消息的平均长度，我们的关键字匹配系统现在比原始的实现要快 10 到 100 倍。

如果你正在做许多正则处理，或者其他的模式匹配，我强烈推荐挖掘下各种不同的前缀和后缀 trie 树的变体，可能会有助于你迅速找到问题的快速解决方案。

12.4.4 报告、监控、调试和部署

我们维护了大量运行于我们的 Python 软件以及其余支撑它的基础设施之上的不同系统。让它保持上线并无中断运行是有难度的。这里有一些我们在途中所学习到的教训。

无论是在你自己的软件中还是在它运行的基础设施上，Python 软件既能实时看到你的系统运行情况，又能看到系统的历史运行情况，真的很强大。我们使用 Graphite 和 `collectd`、`statsd` 一起来画出运行状况的漂亮图表。这带给我们一种观察趋势的方法，并反溯分析问题来找到根本原因。尽管我们还没有设法实现，但是当你具有超出你所能跟踪的度量指标时，Etsy 的 Skyline 作为一种观察异常的方法看起来很巧妙。另一种有用的工具就是 Sentry，一个针对事件日志的大系统，并且跟踪了集群中的机器所产生的异常。

部署可能是痛苦的，无论你用什么来做。我们已经是 Puppet、Ansible 和 Salt 的用户。它们各有优缺点，但是没有一个让一个复杂的部署问题变得魔法般的顺利。

为了维持我们一些系统的高可用性，我们跨地理运行了多个分布式的基础设施集群，让一个系统保持活跃，而其他的作为热备份，通过更新到具有低存活期（TTL）的 DNS 来完成切换。显然那不总是直截了当的，尤其是当你对数据的一致性有强

约束时。幸亏我们没有太受影响，从而使方法相对直截了当。它也提供给我们一种相对安全的部署策略，即更新我们的其中一个备份集群，并在推动那个集群活跃和更新其他集群之前执行测试。

也和所有人一样，我们真的受使用 Docker 所能做的事情的前景所鼓舞。也和很多其他人一样，我们还仅仅处在摸索阶段来探索怎样来使它成为我们部署过程的一部分。无论如何，获得与它的所有二进制依赖以及所包含的系统库一起以轻量级和可重现的方式快速部署我们软件的能力看起来已经近在眼前了。

在服务器层面，有完整的一套例行工具来让生活变得轻松。Monit 为你做监控很强大。Upstart 和 `supervisord` 降低了运行服务的困难。如果你没有使用一个完整的 Graphite 和 `collectd` 设置，Munin 对于一些快速而简单的系统级的图表是有用的。Corosync / Pacemaker 可能对于跨集群节点运行服务是一个良好的解决方案（例如，你有一些需要运行在某处的服务，而不是运行在所有地方）。

我已尝试过在这里不仅是列举出时髦的术语，而且要向你指出我们每天正使用的软件，真正对我们能够有效部署并运行系统产生重大影响。如果你已经全部听说过它们，我确信你肯定有一套完整的其他有用的窍门要分享，所以请给我写信分享几点。如果不是，就去签出它们——希望其中一些对你有用，就像对我们有用一样。

12.5 PyPy 促成了成功的 Web 和数据处理系统

Marko Tasic (https://github.com/mtasic85)

PyPy 是 Python 的一种实现。因为我早期有使用 PyPy 的丰富经验，所以我选择在适用的每处地方都使用它。我从速度关键的小规模玩具类型的项目一直到中等规模的项目都使用过它。我使用它的第一个项目是实现一个协议，我们所实现的协议是 Modbus 和 DNP3。之后，我使用它来实现一个压缩算法，每一个人都惊诧于它的速度。如果我回忆准确的话，我使用的在产品中的第一个版本是 PyPy 1.2，具有开箱即用的 JIT。到 PyPy 的 1.4 版本之前，我们确信它就是我们所有项目的未来，因为许多错误得到了修复，而且速度增加得越来越快。我们好奇于只是通过把 PyPy 更新到下个版本，简单的案例就加快了 2 到 3 倍。

我将解释两个独立但是深度相关的项目，彼此共享了 90%相同的代码，但为了让解释易于接受，我把它们俩统称为“项目”。

该项目就是创建一个系统来收集报纸、杂志和博客，在需要的地方应用 OCR（光学字符识别），把它们进行分类、翻译，应用情感分析，分析文档结构，并为以后

的搜索来做索引。用户能够以任意一种支持的语言来搜索关键词并检索出和索引文档相关的信息。搜索是跨语言的，这样用户就可以用英语来写并以法语来得到结果。此外，用户将从文档页上接收到高亮的文章和关键词，具有关于空间占用和出版价格方面的信息。更高级的使用场景将是报告生成，用户能够看到结果的制表视图，是有关于任何特定的公司在被监控的报纸、杂志和博客上所做广告花费的具体信息。除了广告之外，也能“猜测”一篇文章是付费的还是客观的，并决定它的基调。

12.5.1 先决条件

显然，PyPy 是我们所偏爱的 Python 实现。我们使用 Cassendra 和 Elasticsearch 作为数据库。缓存服务器使用了 Redis。我们使用 Celery 作为一个分布式的任务队列（工作者），使用 RabbitMQ 作为它的经纪人。结果保存在 Redis 的后端。以后，Celery 更加专门地使用 Redis 来作经纪人和后端。所使用的 OCR 引擎是 Tesseract。所用的语言翻译引擎和服务器是 Moses。我们使用 Scrapy 来爬取网站。对于整个系统的分布式锁，我们使用了一个 ZooKeeper 服务器，但是一开始使用的是 Redis。Web 应用基于优秀的 Flask Web 框架以及它的许多扩展，例如 Flask-Login、Flask-Principle 等。Flask 应用由每台 Web 服务器上的 Gunicorn 和 Tornado 所承载，nginx 用来作为 Web 服务器的反向代理。代码的其余部分由我们自己来写，是运行在 PyPy 之上的纯粹的 Python。

整个项目搭建于公司内部的 OpenStack 私有云上，并取决于需求，执行了 100 到 1000 个 ArchLinux 实例，能够在线动态调整。整个系统每 6 到 12 个月消耗 200 TB 的存储，取决于所提到的需求。除了 OCR 和翻译之外，所有的处理由我们的 Python 代码来完成。

12.5.2 数据库

我们为 Cassandra、Elasticsearch 和 Redis 开发了具有统一模型的类的 Python 包。它是一个简单的 ORM（对象关系映射），在许多条记录要从数据库来获取的情况下，把每样东西映射成一个字典或者字典的列表。

既然 Cassandra 1.2 不支持在索引上做复杂的查询，我们用类似 join 的查询来支持它们。无论如何，我们允许小规模数据集上的复杂查询（直到 4GB 为止），因为许多东西必须要放在内存中处理。PyPy 运行在那些 CPython 甚至无法把数据装载进内存的场景下，多亏了它应用同构列表的策略，从而使得它们在内存中更加紧凑。PyPy 的另一个好处就是它的 JIT 编译在发生数据操作或分析的循环中开始运转。我们以这样一种方式来写代码，那就是类型在循环内部保持静态，因为在那里 JIT 编译的代码尤其良好。

Elasticsearch 被用来做索引以及文档的快速检索。当涉及查询复杂性时，它非常灵活，这样我们使用它就不会产生主要的问题。我们拥有的其中一个问题和文档更新有关，它不是设计用来快速修改文档的，这样我们不得不把那部分迁移到 Cassendra。另一个限制和侧面（facet）以及数据库实例所需要的内存有关，但是通过产生更多更小的查询，然后手动操纵在 Celery 工作者中的数据，问题得到了解决。在 PyPy 和被用来与 Elasticsearch 服务器池做交互的 PyES 库之间没有浮现出主要的问题。

12.5.3 Web 应用

就如上面所提到的那样，我们使用 Flask 框架以及它的第三方扩展。初始阶段，我们用 Django 来开始所有的工作，但是由于需求的快速改变，我们切换到了 Flask。这并不意味着 Flask 比 Django 更好，它对我们来说只是用 Flask 比用 Django 更容易来跟踪代码，因为它的项目布局非常灵活。Gunicorn 被用作一个 WSGI（Web 服务器网关接口）的 HTTP 服务器，它的 IO 循环由 Tornado 来执行。这允许我们让每个 Web 服务器达到 100 个并发连接。这比所期望的要低，因为许多用户查询可能花费较长的时间——用户的请求产生了许多分析，数据以用户交互的方式来返回。

初始阶段，Web 应用依赖于 Python 映像库（PIL）来做文章和单词的高亮。我们一起使用 PIL 库和 PyPy 时发生了问题，因为那时 PIL 有许多内存泄漏。接着我们切换到了 Pillow，它维护的频率更加高。最后，我们通过 subprocess 模块写出了与 GraphicsMagick 做交互的库。

PyPy 运行良好，结果和 CPython 兼容。这是因为通常 Web 应用是 IO 密集型的。无论如何，随着 PyPy 中 STM 的开发，我们希望不久之后就有在多核实例层面上的可扩展的事件处理。

12.5.4 OCR 和翻译

我们为 Tesseract 和 Moses 写了纯粹的 Python 库，因为我们在使用依赖于 CPython API 的扩展时发生了问题。PyPy 在使用 CPyExt 时对 CPython API 具有良好的支持，但是我们想要对藏在表面下的所发生的事情具有更多的控制力。结果就是，我们制作了一个兼容 PyPy 的解决方案，具有比在 CPython 上运行稍微快一点的代码。它没有更快的原因就是大多数处理发生于 Tesseract 和 Moses 的 C/C++代码中。我们只能加速输出处理以及 Python 结构文档的构建。在这个阶段没有主要的 PyPy 兼容性问题。

12.5.5 任务分发和工作者

Celery 带给我们在后台运行许多任务的力量。典型的任务是 OCR、翻译、分析等。所有的事情都能用 Hadoop 的 MapReduce 来做，但是我们选择了 Celery，因为我们知道项目需求可能经常变更。

我们有大概 20 个工作者，每个工作者有 10 到 20 个函数。几乎所有的函数都有循环，或者有许多嵌套的循环。我们小心地让类型保持静态，这样 JIT 编译器就能显身手了。最后的结果就是以 2 到 5 倍的加速超过了 CPython。我们没有得到更好的速度提升的原因是我们的循环相对较小，在 2 万到 10 万次迭代之间。在有些我们必须在单词层面上做分析的情况下，我们具有超过 1 百万次迭代，这就是我们得到超过 10 倍的速度提升的地方。

12.5.6 结论

PyPy 对于每一个依赖于可读、可维护的大型源码的执行速度的纯 Python 项目而言是一个优秀的选择。我们发现 PyPy 也十分稳定。我们的所有程序都是长期运行的，使用静态和/或在数据结构内部的同构类型，这样 JIT 就能显身手了。当我们在 CPython 上测试整体项目时，结果并没有让我们吃惊：我们用 PyPy 比 CPython 大概具有 2 倍的速度提升。在我们的客户眼里，这意味着以相同的价格得到了 2 倍更好的性能。除了 PyPy 迄今为止带给我们的所有好处，我们希望它的软件事务内存（STM）的实现将带给我们可扩展地来并行执行 Python 代码。

12.6 在 Lanyrd.com 中的任务队列

Andrew Godwin (lanyrd.com)

Lanyrd 是一家发现社交协会的网站——我们的客户登入后，我们使用来自社交网络的他们的好友图，以及其他类似于他们的工作行业或者地理位置之类的线索来建议相关的协会。

网站的主要工作就是提取出原始数据的精华，这样我们就能展现给用户——尤其是一个排序好的协会列表。我们必须要离线来做，因为我们每隔几天刷新推荐的协会列表，也因为我们遇到了通常较慢的外部 API。我们也为其他花费较长时间的事情使用 Celery 任务队列，比如获取人们所提供的链接的缩略图以及发送电子邮件。每天在队列中通常有超过 100 000 个任务，有时会更多。

12.6.1 Python 在 Lanyrd 中的角色

Lanyrd 从一开始起就由 Python 和 Django 来构建，几乎它的每一部分都是用 Python 来写的——网站本身，离线处理，我们的统计和分析工具，我们的移动后端服务器以及部署系统。它是一种多用途和成熟的语言，相当容易快速用它来写代码，最感谢的就是大量可用的库以及语言良好的可读性和简洁的语法，这意味着容易更新和重构，并且一开始也容易编写。

当我们对任务队列的需求发生演进（很早就开始）时，Celery 任务队列已经是一个成熟的项目了，Lanyrd 的其余部分已经使用了 Python，所以它自然就适应了。当我们规模变大时，有一个需求要改变支撑它的队列（最后用了 Redis），但它一般有很好的扩展性。

作为一个初创公司，为了取得进展，我们不得不欠下一些已知的技术债——这只是你不得已做的，当问题可能浮现时，只要你知道它们是什么问题，就不一定是坏事。Python 在这方面的灵活性非常棒，它一般鼓励组件之间的松耦合，这意味着发布一些“足够好”的实现常常是容易的，然后将来轻易地重构出一个更好的实现。

任何关键性的东西，例如付款代码，要有完整的单元测试覆盖率，但是对于网站的其他部分和任务队列流（尤其和显示相关的代码），往往前进得太快，让单元测试变得没有价值（它们太脆弱）。取而代之的是，我们采用了一个非常敏捷的方法，具有一个短达 2 分钟的部署时间以及优秀的错误追踪。如果出现错误，我们常常能够修复它，并在 5 分钟内完成部署。

12.6.2 使任务队列变高性能

任务队列的主要问题是吞吐量。如果它有任务积压，那么网站继续工作但是开始变得有些不可思议的延时——列表没有更新，页面内容是错的，电子邮件几小时都没发送出去。

然而，幸运的是，任务队列也鼓励非常具有扩展性的设计，只要你的消息中心服务器（在我们的案例中是 Redis）能够处理任务请求和响应的消息开销，对于实际的处理而言，你可以运转任意数量的守护工作进程来应对负载。

12.6.3 报告、监控、调试和部署

我们有跟踪队列长度的监控，如果它开始变长，我们只要部署另一台有更多守护工作进程的服务器即可。Celery 让这样做变得很容易。我们的部署系统有钩子，通过它，我们能够在盒子上增加工作线程的数量（如果我们的 CPU 利用率不是最优的），

并且能够在 30 分钟内轻易地把一台全新的服务器转变成一个 Celery 工作者。它和降到最低水平的网站响应时间不一样——如果你的任务队列突然得到一个负载高峰，你有一些时间来实现一个修复，如果你留下了足够的冗余容量，通常它能平滑下来。

12.6.4 对开发者同行的建议

我的主要建议将会是尽可能快速地把尽量多的任务塞到任务队列中去(或者一个类似的松耦合的架构)。它在初始阶段要花费一些工程方面的努力，但是当你规模扩大时，曾经要花半秒时间做的操作可能增大到半分钟才能完成，你会高兴于它们没有阻塞你的主渲染线程。一旦你走到了这一步，要确保你紧密监控你的平均队列延迟（一个任务从提交到完成需要花多久），并且确保当你的负载增加时，还有一些冗余的容量。

最后，要注意为多个不同优先级的任务配置多个任务队列是有意义的。发送电子邮件不具有很高的优先级，人们习惯于电子邮件过几分钟才到达。无论如何，如果你在后台渲染缩略图并且当你正在做时，显示一个旋转图标，你想要让这种工作成为高优先级，因为如果不是这样的话，你会产生糟糕的用户体验。你不想在接下来的 20 分钟内在你的网站上让 100000 人的邮件广告全都延迟显示缩略图。

作者简介

Micha Gorelick 在 bitly 公司从事与数据打交道的工作，并以负责建立了快速前进实验室（Fast Forward Labs），研究从机器学习到高性能流算法领域的问题。

Ian Ozsvald 是一个数据科学家，并且在 ModelInsight.io 担任 Python 老师，具有超过 10 年的 Python 经验。他已经在 PyCon 和 PyData 大会上讲课超过 10 年，并且在伦敦从事人工智能和高性能计算领域的咨询工作超过 10 年时间。Ian 的背景涉及 Python 和 C++，结合了 Linux 和 Windows 开发、存储系统、许多自然语言处理和文本处理，机器学习以及数据可视化。他在许多年前也共同创建了专注于 Python 的视频学习网站 ShowMeDo.com。

封面简介

本书封面上的动物是矛头蝮蛇。在法语字面上是“钢矛”的意思，该名字为一些主要发现于马提尼克岛上的蛇的种类所保留。它也可能用来指其他的矛头蝮蛇种类，比如圣卢西亚矛头蝮蛇、普通矛头蝮蛇，以及三色矛头蝮蛇。所有这些种类都属于蝮蛇，所以就以两个位于眼睛和鼻孔之间的看起来像斑点的热感应器官来命名。

由于大部分三色矛头蝮蛇和普通矛头蝮蛇能导致咬伤后致死，所以在美洲矛头蝮属种的蛇比任何其他的属种要为更多人的死亡而负责。在南美咖啡和香蕉种植场的工人惧怕被意图捕获啮齿类动物当点心的普通矛头蝮咬一口。当你在中美洲的河流岸边沐浴在阳光下而不堪忍受寂寞的生活时，要当心据说脾气更暴躁的三色矛头蝮蛇，它是一种危险。

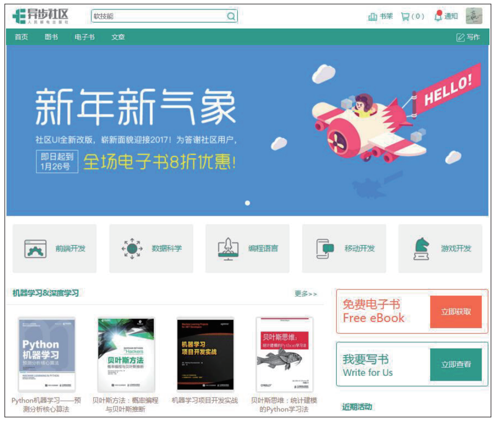
O’Reilly 封面上的许多动物是濒危物种，它们对这个世界是重要的。为了了解到更多关于你该如何去帮助它们的信息，请去 animals.oreilly.com。

封面图片来自于 Wood 的动画创造。封面字体是 URW 打字机字体和 Guardian Sans 字体，文本字体是 Adobe Minion Pro 字体，标题是 Adobe Myriad Condensed 字体，代码字体是 Dalton Maag 的 Ubuntu Mono 字体。

欢迎来到异步社区！

异步社区的来历

异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。



异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题 可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户账户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在 

里填入可使用的积分数值，即可扣减相应金额。

特 别 优 惠

购买本书的读者专享**异步社区购书优惠券**。

使用方法：注册成为社区用户，在下单购书时输入 57AWG 使用优惠码，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得 100 积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：

异步社区	微信服务号	微信订阅号	官方微博	QQ 群：436746675

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@ 人邮异步社区，@ 人民邮电出版社 - 信息技术分社

投稿 & 咨询：contact@epubit.com.cn

O'REILLY®

Python高性能编程

Python代码仅仅能够正确运行还不够，你需要让它运行得更快。通过探索设计决策背后的基础理论，本书帮助你更加深刻地理解Python的实现。你将学习如何找到性能瓶颈，以及如何在大数据量的程序中显著加快代码。

如何利用多核架构或集群的优点？如何构建一个在不损失可靠性的情况下具备可伸缩性的系统？有经验的Python程序员将学到针对这些问题或者其他问题的具体解决方案，以及来自那些在社交媒体分析、产品化机器学习和其他场景下使用高性能Python编程的公司的成功案例。

通过阅读本书，你将能够：

- **更好地掌握numpy、Cython和剖析器；**
- **了解Python如何抽象化底层的计算机架构；**
- **使用剖析手段来寻找CPU时间和内存使用的瓶颈；**
- **通过选择合适的数据结构来编写高效的程序**
- **加速矩阵和矢量计算；**
- **使用工具把Python编译成机器代码；**
- **管理并发的多I/O和计算操作；**
- **把多进程代码转换到在本地或者远程集群上运行；**
- **用更少的内存解决大型问题。**

“尽管Python在学术和工业领域很流行，但人们也经常由于Python程序运行太慢而放弃它。本书通过全面介绍改善优化Python计算速度和可扩展性的策略，从而消除人们的这种误解。”

——Jake VanderPlas
华盛顿大学

Micha Gorelick在bitly公司从事与数据打交道的工作，并负责建立了快速前进实验室（Fast Forward Labs），研究从机器学习到高性能流算法领域的问题。

Ian Ozsvald是ModelInsight.io的数据科学家和教师，有着超过十年的Python经验。他在PyCon和PyData会议上教授Python编程，这几年一直在英国从事关于数据科学和高性能计算方面的咨询工作。

PYTHON / PERFORMANCE

异步社区 人民邮电出版社 www.epubit.com.cn
异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

封面设计：Karen Montgomery 张健

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-45632-8

分类建议：计算机／软件开发／Python
人民邮电出版社网址：www.ptpress.com.cn